Department of Informatics

A comparison of information modeling in ORM and OWL

Master's Thesis

Charlotte Løvdahl

November 4, 2011

Acknowledgments

Kjell, for your patience and for being logical and rational when I was not;

Ellen and Martin, for your guidance and help with arrows, boxes and sparkling owls;
Ellen, for weekly meetings, constant feedback and for using a pencil instead of a red pen;
Arild Waaler, for connecting me to the Norwegian National Register;

Friends and fellow students on the 7th floor@IFI, for the coffee, hugs and pep talks;
For all of you who helped me spell check, you know who you are;
My loving and caring family, for always encouraging and supporting;

Thank you all!

Abstract

The Semicolon II project looks at ways to enhance the interoperability between Norwegian government institutions by using metadata and conceptual models. The Web Ontology Language (OWL) has been chosen to create the metadata models for these institutions. This thesis focuses on comparing Web Ontology Language (OWL) and Object Role Modeling (ORM). ORM is a well established modeling language in comparison to OWL which is relatively young. Their capability and suitability to create information models are investigated with a mapping from ORM to OWL. This mapping revealed that OWL can construct most of the constraints in ORM, except for the Equivalence-Of-Path constraint. The Open World Assumption in OWL complicates information modeling by not forcing restrictions on classes. This allows for incomplete information: information which is expected, but not found, is stated as unknown. Thus a model can violate the rules on which it is built. How to deal with this disadvantage is a topic for future research. The languages ability to support and underpin the information systems they model are investigated by researching tools that combine data from databases with the models. There are, to our knowledge, no tools for ORM which allows its models to connect to data in a database, while OWL has several. Based on a survey of available tools, we have chosen to apply the tool D2RQ. The result is an OWL model with a bridge to the data in the database. This allows the information system to use the database's strengths on query performance and storage. In addition the populated model can be reasoned over and put on the Web for easy sharing.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The Semicolon II project (Semicolon II, 2011) is a cooperation between research and computer science companies, government institutions of Norway and Nordic universities. The main goal of the project is to simplify the exchange of information between government institutions of Norway. It is created as a response to the government institutions' desire to provide better service to businesses and citizens alike. They aim to do this by improving interoperability which can enhance the collaboration across the public sector as a whole. One of the approaches is to investigate how metadata and conceptual models based on the different institutions can be used to enhance the flow of information between the institutions.

Ontologies are a means to formally represent knowledge within a domain. An ontology is defined as a representation of the human meaning of terms in a vocabulary and the relationships between these terms (McGuinness, Van Harmelen, et al., 2004). In the Semicolon II project ontologies can be used to create the metadata and conceptual models of the different institutions.

The Web Ontology Language (OWL) is a language for defining ontologies and is declared the standardized language for the Semantic Web by the World Wide Web Consortium (W3C) (Hebeler, Fisher, Blace, & Perez-Lopez, 2009). It has its origin in the field of Artificial Intelligence. OWL is better suited to create the ontologies for the Semicolon II project than other formalisms, such as the Unified Modeling Language, because it has a vocabulary that can be used to share the understanding and meaning of terms that are common for the institutions. The Semantic Web is a Web of data tagged with meaning for increased usability (Lee, Hendler, Lassila, et al., 2001) so models written in OWL can label data with information that is meaningful for humans and applications. OWL models are possible to put on the Web for easy access and even more important, they can be reasoned over. Reasoning makes it possible to infer more information than what is explicitly stated in the model and add this information as it is derived, which leads to implicit knowledge being made available and enhances its usability (Hebeler et al., 2009).

Accessibility combined with reasoning makes OWL a very suitable language for creating information models. An alternative to OWL is the modeling language Object Role Modeling (ORM). It creates information models about a certain domain, or a Universe of Discourse, and has its origin from the field of databases (Halpin, Morgan, & Morgan, 2008). ORM models elementary facts about the world with precision which gives simplicity, stability, ease of validation and makes it easy to perform conceptual analysis.

It is desired to compare two aspects of these languages. The first aspect is the languages capabilities. How capable and suitable are they to create information models? The second aspect is their abilities to underpin and support information systems. Are there any tools for the languages that allow them to connect the data in the database to the information model?

To investigate the first problem we compare the most important parts of the expressivity of the two languages. This can be done by performing a mapping between the two languages, to check if the basic structures in one language can be translated to the other. Mapping is done by following rules that state how a structure in one language can be translated to another language. It is interesting to perform a mapping from an ORM model to an OWL model since the OWL language is fairly young and is still being refined, while the ORM language has been around for some time and is well established as a modeling language. The mapping was done manually since, to the best of our knowledge, there are no tools to do it automatically.

We chose to use the Norwegian National Register, one of the government institutions that are part of the Semicolon II project, as a basis for the ORM model. To demonstrate the basic and most frequently used structures that exist in an ORM model a simplified version of the Norwegian National Register was created which contains the most important parts of the ORM language. This model was mapped to an OWL model.

The second problem requires a research to find existing tools for the languages that can connect to databases. It is well known that the ORM language has a Relational Mapping Procedure which creates database schemas from an information model and that it is included in ORM modeling tools. The drawback is that even though ORM can create database schemas it loses all connections to the schema and the database after its creation. Since it is unknown to us whether or not the OWL language has anything similar, it is necessary to conduct a research on methods and tools available. Are there tools for the OWL language that allows the OWL model to be used together with a database? One type of tools that creates a connection from databases to ontologies is mapping tools.

A research was conducted to find tools that performs mapping from a database to an OWL model. It was required that the tools are, among other things, available, free of charge and user friendly. In addition it was desired to find different ways of dealing with the data in the database. Two approaches was discovered, one where the tools extract the data from the database and creates a populated ontology, and one where bridges was created from the ontology to the database so the data still resides in the database.

After having obtained an overview of the available tools and compared them to each other based on the accessible literature about them, the tool D2RQ was chosen for a closer examination. D2RQ is free to download, user friendly and the resulting model can be queried and easily shared as it can be put on the Web.

During the early phase of the comparison of the languages it was thought that ORM and OWL are rather similar considering all the features they have in common. It was discovered in the mapping process that there are in fact some very fundamental differences between them that have major consequences for their modeling abilities.

The rest of the thesis is structured as follows: Chapter 2 gives a brief introduction to the two modeling languages ORM and OWL in addition to other necessary background information. Chapter 3 provides a more detailed description of the problems and the requirements for solving them. Chapter 4 gives an overview over the existing tools found during the research

that performs mapping from a database to an ontology. Chapter 5 describes the mapping rules used for mapping from ORM to OWL. Chapter 6 shows the ORM model created of the Norwegian National Register and how it was mapped to an OWL model. Chapter 7 describes how mapping from a database to an OWL model is done by the tool D2RQ and how the result can be used. The lessons learned are discussed in Chapter 8. Finally in Chapter 9 we conclude and discuss future work.

The models, mappings and other documents created during this thesis, are available at the Web site `http://sws.ifi.uio.no/project/dsf/`.

Chapter 2

Background

This chapter provides an introduction of the modeling languages Web Ontology Language (OWL) and Object Role Modeling (ORM). The first section gives a quick introduction of the language OWL, followed by a section with an introduction of the language ORM. The last section is an overview of the main differences between the two languages.

2.1 The Web Ontology Language

This section as a whole refers to the works by Hebeler et al. (Hebeler et al., 2009) and Hitzler et al. (Hitzler, Krötzsch, & Rudolph, 2009). The paragraphs about the Semantic Web refers in addition to the work by Berners-Lee (Lee et al., 2001) and the paragraphs about the constraints refer in addition to the work by Bechhofer et al. (Bechhofer et al., 2004).

2.1.1 The Semantic Web

The vision of the Semantic Web is that one day it will be an extension of the World Wide Web (WWW) with intertwined information from different sources, tagged with meaningful labels. The goal is to weave together the already existing enormous network of human knowledge by creating globally accessible information, which can be exchanged and used based on the exploitation of machine-processable metadata (Davies, Fensel, & Van Harmelen, 2003). The Semantic Web wishes to increase the utility of the information so that computers and applications everywhere can improve searching and browsing, be able to automatically check information, come to reasonable conclusions, and maybe even ”think”.

Reasoning is a mean to help computers and applications achieve this. Reasoning can derive knowledge from different sources which allows it to infer and add knowledge that is only implicitly stated. To allow data to be added to the Semantic Web bit by bit, reasoning need to comply with the Open World Assumption (OWA). OWA makes the assumption that any knowledge base is incomplete, meaning that if something is not stated, it is not so that it does not exist, it is merely *unknown*. It is unknown because it may exist, but simply has not been added yet. To structure the knowledge the Semantic Web uses ontologies which is the representation of the meaning of terms in a vocabulary and the relationships between these

terms (McGuinness et al., 2004). The de facto language for making ontologies for the Semantic Web is the Web Ontology Language (OWL).

2.1.2 The Web Ontology Language

The Web Ontology Language (OWL) is a World Wide Web Consortium (W3C) approved standardized modeling language for the Semantic Web (McGuinness et al., 2004). An OWL ontology is a Resource Description Framework (RDF) graph which is made up by a set of statements. Statements, also called triples, consist of three parts, a subject, a property and an object. Graphically it is constructed like a directed graph where the subjects and objects are nodes and the properties are edges. Figure 2.1 shows a graph with two triples. The ellipse to the left with `Diana` is the subject of these triples, the arrows called `likes` and `hasName` are properties and the second ellipse with `Kim` and the rectangle `"Diana"` are objects. These triples are statements that tell something about the subject `Diana`. They can be formulated as `Diana - likes - Kim` and `Diana - hasName - "Diana"`.

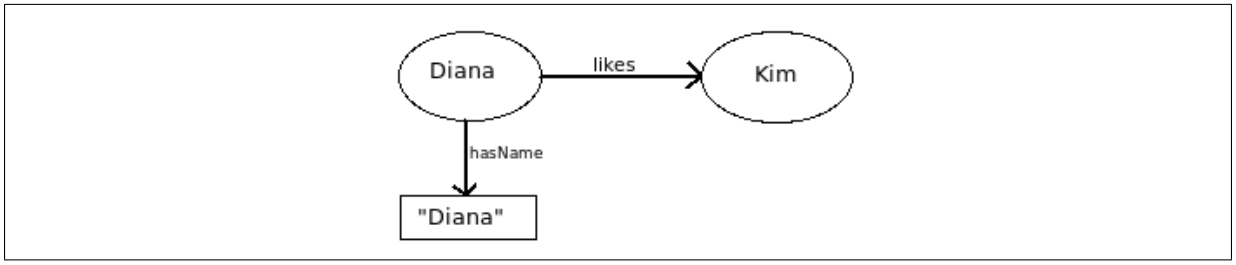


Figure 2.1: A graph with two triples consisting of a common subject, two properties and two objects.

OWL is able to perform reasoning on triples because it is based on Description Logic (DL). DL is a fragment of first-order predicate logic and a formalism for representing knowledge. OWL was designed in DL in order to achieve a beneficial tradeoff between expressivity and scalability, while at the same time maintaining the possibility to reason over the logic. In DL the different logic axioms and assertions are put in boxes, called T-box and A-box . T-box stands for terminological knowledge and holds the definitions and ”the semantics” in an ontology, since it tells how classes and properties are related to one another. The terminology box is independent from the actual data in the ontology. A-box stands for assertional knowledge and contains the actual data, the facts about concrete individuals such as a, b and c. The assertion box also contains sets of class membership assertions, C(a), and property assertions R(a, b).

Classes, Properties and Individuals

The basic building blocks in OWL are classes, properties and individuals. A *class* is a kind of resource that represents a set of several resources that share common characteristics or are similar in other ways. OWL has a predefined class called `owl:Thing` and all individuals are members of this superclass. An *individual* is a resource that is a member of a class and represents an instance of that class. A *property* is used to describe a resource, and it is itself a resource. It is used to establish relationships between resources in a certain direction, from subject to object. A property can be divided into two main types, object properties and datatype properties. Object properties are used to connect, or link, individuals to other individuals, as seen in the

triple in Figure 2.1 with the object property `likes`. Datatype properties on the other hand link individuals to literal values as seen in Figure 2.1 with the datatype property `hasName`. Classes and properties can exist separately of each other, and properties are not bound to a specific class unless explicitly stated.

When thinking about classes and properties in OWL, it helps to think in terms of sets. In Figure 2.2 there is an example with two classes, class `A` and class `B` which consists of several resources, or individuals. In the figure the resources are represented as dots. One of the resources in class `A` has a property `r1`, represented by the directed arrow, to a resource in class `B`.

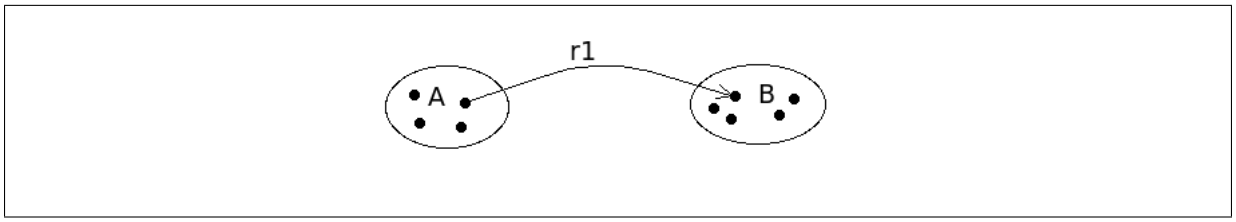


Figure 2.2: The set A and set B, with a property r1 linking a resource from set A to a resource in set B.

Domain and Range

In the example in Figure 2.2 the classes `A` and `B` can be connected through the property `r1` by using `rdfs:domain` and `rdfs:range`. Domain and range describes the direction of the relationship between resources, from subject to object. `rdfs:domain` specifies the type of all individuals who are the subject of statements that contain the property being described. `rdfs:range` specifies the type of all individuals or datatypes that are the object of statements that contain the property being described. Range can also specify valid literals to be used as objects in statements. It is important to notice that `rdfs:range` is a global restriction so its restriction will apply to the property regardless of the class to which the property is applied. In Figure 2.3 the object property called `hasLastName` has a domain `Person` and a range `LastName`. The property `hasLastName` must in all triples where it is used, have an individual from class `Person` as a subject of the triples and individuals that are member of class `LastName` as object. Object properties can be stated as inverse of each other by using the property `owl:inverseOf`. This allows the domain and range for statements to be exchanged, so the relationship can exist in the opposite direction. In Figure 2.3 the inverse property of the object property `hasLastName`, is `isLastNameFor`, which automatically gets class `LastName` as domain and class `Person` as range. It is not necessary to explicitly state it. For this chapter and all the following, all examples of OWL code will be written in Turtle syntax.

```
:hasLastName rdf:type owl:ObjectProperty ;
 rdfs:domain :Person ;
 rdfs:range :LastName ;
 owl:inverseOf :isLastNameFor .

:isLastNameFor rdf:type owl:ObjectProperty .
```

Figure 2.3: The object property hasLastName with domain and range.

Relations between Classes

OWL classes can also be put in relation to each other by using the properties `rdfs:subClassOf` and `owl:equivalentClass`. If class `B` is a subclass of class `A`, then the set of individuals in class `B` should be a subset of the set of individuals in class `A`. A class is by definition a subclass of itself as the subset may be the entire complete set. On the other hand, if class `B` is equivalent to class `A`, the two classes will be treated as a single resource. All class restrictions and class extensions are then shared between `B` and `A`. The two classes will contain exactly the same set of individuals. The property `owl:equivalentClass` can also be used to define a class. It can be used to state necessary conditions for class membership. An individual called `d` can become a member of class `A` if the individual fulfills the conditions.

The properties `rdfs:subClassOf` and `owl:equivalentClass` can also be used to state a relationship to another class by using property restrictions. A property restriction is a special kind of class used to describe an anonymous class of type `owl:Restriction` that applies for all individuals that satisfy the restriction. It has a property `owl:onProperty` that links the restriction to a certain property. Anonymous classes (and individuals) are written in square brackets [] in Turtle. In Figure 2.4 the class `Person` has a property restriction on the property `hasGender` so that it needs to have all of its values from the class `Gender`. A property restriction in a class applies for all individuals in that class, so this restriction applies for all people. Notice that this property restriction is redundant if the range of `hasGender` is the class `Gender`, since all the values for this property will have to be from the class `Gender` anyway. A property restriction can not violate the range of the property.

```
 :Person rdf:type owl:Class ;
   rdfs:subClassOf
[ rdf:type owl:Restriction ;
  owl:onProperty :hasGender ;
  owl:allValuesFrom :Gender ] .
```

Figure 2.4: An OWL class with a property restriction.

In Figure 2.4 the class `Person` is a subclass of the property restriction which means that `Person` (the set of all individuals in the class `Person`) is a subclass of all individuals that have the property `hasGender` with the value from class `Gender`. It does not prevent the class `Cat` from having the property `hasGender`, it simply states that for all individuals in class `Person` that have this property, the value of it need to be an individual from class `Gender`. This is different from using `owl:equivalentClass`. If the statement is rewritten with `owl:equivalentClass` instead of `rdfs:subClassOf` it will state that all individuals that have the property `hasGender` with the value from class `Gender` are members of the class `Person`.

There are two kinds of property restrictions, namely value constraints and cardinality constraints. In order to explain how constraints work it helps to think in terms of triples. The restrictions are on the triples that uses the property and restricts either the number of triples, the values allowed in the object of these triples, or both.

Value Constraints

A value constraint puts constraints on the range of a property *when applied to this particular class description*. Note that this is different from `rdfs:range`, which is applied to all situations in which a property is used. Value constraints restrict the values in the object of triples where the property is used. There are three kinds of value constraints: `owl:allValuesFrom`, `owl:someValuesFrom` and `owl:hasValue`.

owl:allValuesFrom

The value constraint `owl:allValuesFrom` is a property that restricts the object of all triples to be either a member of the class stated or a data value within the specified data range. As seen in Figure 2.5, `owl:allValuesFrom` applies for all the individuals in the class `DogOwner` that have triples with the property `hasPet`. The constraint restricts the object of all these triples has to be an individual from the class `Dog`.

```
 :DogOwner rdf:type owl:Class ;
   rdfs:subClassOf
[ rdf:type owl:Restriction ;
  owl:onProperty :hasPet ;
  owl:allValuesFrom :Dog ] .
```

Figure 2.5: An OWL class with an owl:allValuesFrom restriction.

owl:someValuesFrom

The value constraint `owl:someValuesFrom` is a property that request the present of *at least* one triple, where the value of the object is either a member of the class stated or a data value within the specified data range. This constraint does not exclude that there may be other triples where the object are not members of the class or a data type within the data range. In Figure 2.6 the constraint `owl:someValuesFrom` applies for all individuals in the class `Garden` that have triples with the property `hasFlowers`. The constraint request that at least one of the triples has an object with a value from class `Rose`. There may be other triples with values from other classes.

```
 :Garden rdf:type owl:Class ;
   rdfs:subClassOf
[ rdf:type owl:Restriction ;
  owl:onProperty :hasFlowers ;
  owl:someValuesFrom :Rose ] .
```

Figure 2.6: An OWL class with an owl:someValuesFrom restriction.

owl:hasValue

The value constraint `owl:hasValue` is a property that requests the presence of *at least* one triple where the value of the object is semantically equal to the value stated. In Figure 2.7 the constraint `owl:hasValue` requests that for all individuals of class `GoodPeople` there has to be at least one triple with the property `hasQuality` where the value of the object is `friendly`. There may be other triples with other values in the object.

```
 :GoodPeople rdf:type owl:Class ;
   rdfs:subClassOf
[ rdf:type owl:Restriction ;
  owl:onProperty :hasQuality ;
  owl:valuesFrom :friendly ] .
```

Figure 2.7: An OWL class with an owl:valuesFrom restriction.

Qualified Cardinality Constraints

A qualified cardinality constraint combines value restrictions with cardinality restrictions. This restriction makes it possible to specify the number of triples for each individual in a class, where a certain property is used, and the value of the object. The object has to be an individual of the specified class or a value within the specified data range. The number of different triples is, in the same way as cardinality constraints, counted as the number of semantically distinct values in the object of the triple. There are three types of qualified cardinality constraint; `owl:maxQualifiedCardinality`, `owl:minQualifiedCardinality`, `owl:qualifiedCardinality`.

owl:maxQualifiedCardinality

The qualified cardinality constraint `owl:maxQualifiedCardinality` expresses that there need to be *at most N* properties that each point to an individual in the specified class, or a datatype within the specified data range. In Figure 2.8 individuals of class `Bus` can have no more than `50` properties `hasPassenger` with a value from class `Person`.

```
 :Bus rdf:type owl:Class ;
   rdfs:subClassOf
[ rdf:type owl:Restriction ;
  owl:onProperty :hasPassenger ;
  owl:onClass :Person ;
  owl:maxQualifiedCardinality "50"^^xsd:nonNegativeInteger ] .
```

Figure 2.8: An OWL class with an owl:maxQualifiedCardinality restriction.

owl:minQualifiedCardinality

The qualified cardinality constraint `owl:minQualifiedCardinality` expresses that there need to be *at least N* properties that each point to an individual in the specified class, or a datatype within the specified data range. In Figure 2.9 all individuals of class `Parent` must have at least `1` property `hasChild` with a value from class `Child`.

```
  :Parent rdf:type owl:Class ;
   rdfs:subClassOf
[ rdf:type owl:Restriction ;
  owl:onProperty :hasChild ;
  owl:onClass :Child ;
  owl:minQualifiedCardinality "1"^^xsd:nonNegativeInteger ] .
```

Figure 2.9: An OWL class with an owl:minQualifiedCardinality restriction.

owl:qualifiedCardinality

The qualified cardinality constraint `owl:qualifiedCardinality` expresses that there need to be *exactly N* properties that each point to an individual in the specified class, or a datatype within the specified data range. In Figure 2.10 all individuals of class `Child` must have exactly `2` properties `hasParent` with a value from class `Parent`.

```
  :Child rdf:type owl:Class ;
   rdfs:subClassOf
[ rdf:type owl:Restriction ;
  owl:onProperty :hasParent ;
  owl:onClass :Parent ;
  owl:cardinality "2"^^xsd:nonNegativeInteger ] .
```

Figure 2.10: An OWL class with an owl:qualifiedCardinality restriction.

Keys

Classes can have keys. They are similar to primary keys in databases and can be used to uniquely identify a member of a class. Keys can consist of one or more properties, so a single property or a combination of several properties can be made unique for a class. When a property in OWL is made into a key it becomes both functional and inverse functional, stating that a member of a class can not have more than one value for the property and two members can not have the same value. To create a key for a class the property `owl:hasKey` is used and the value of that property is the key. Figure 2.11 shows how a class `Car` is given a unique key with the property `hasRegistrationNumber`.

```
:hasRegistrationNumber rdf:type owl:ObjectProperty ;
 rdfs:domain :Car ;
 rdfs:range :RegistrationNumber .

:Car rdf:type owl:Class ;
owl:hasKey ( :hasRegistrationNumber ) .
```

Figure 2.11: An OWL class with a key.

Property Chains

In OWL it is possible to construct property chains which can be reasoned over by the reasoner Pellet (Pellet 2.0 RC6) (Blace, 2009). This construction makes it possible to create a chain of properties from one instance to another, and have that property chain infer that there must be another property from the first instance to the second. The left model in Figure 2.12 shows a directed graph with the classes `Child`, `Parent` and `Brother`. From an individual of class `Child` there is a property `hasParent` that links to an individual of class `Parent`, and a property `hasBrother` that links to an individual in class `Brother`. There is a property chain from the individual of class `Child` to the individual of class `Brother` through `hasParent` and `hasBrother`. By using this chain a reasoner can infer that there must be a property `hasUncle` directly from the individual in `Child` to the individual in `Brother`. The power of a property chain is the ability to infer new information based on the chain. Notice that the inference does not work the other way around, the implication only goes in one direction. The right model in Figure 2.12 has only the property `hasUncle` from an individual in `Child` to an individual in `Brother`. It is not possible to deduct the properties `hasBrother` and `hasParent` (Blace, 2009).

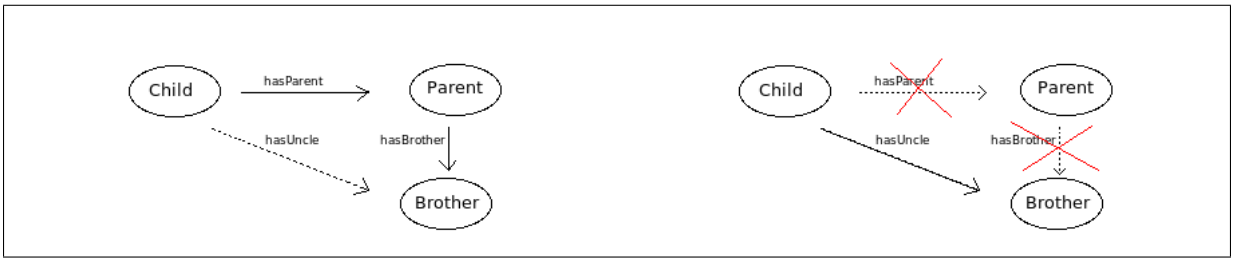


Figure 2.12: The left figure shows how a property chain is used to infer new knowledge. The right figure shows that the implication only goes in one direction.

OWL Document

OWL has no standard graphic notation so it is declared textually. An OWL document in Turtle contains prefixes and namespaces located at the top, followed by the triples that describe the ontology. A namespace is where a vocabulary is defined, for example RDF has the namespace `http://www.w3.org/1999/02/22-rdf-syntax-ns#`. In some syntax, like Turtle, a namespace is referred to by a prefix. In the case of RDF, the prefix is usually `rdf` and written `@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>`. If one wishes to use a term in an ontology, for example `type`, which is defined in RDF, one writes `rdf:type`. The top of the document may contain further information about the ontology such as version and comments.

2.2 Object Role Modeling

This section as a whole refers mostly to the work by Halpin et al. (Halpin et al., 2008), and the paragraph about equivalence-of-path refers to the work by Skagestein and Normann (Skagestein & Normann, 2008; Skagestein, 1996).

2.2.1 Database

A database is a collection of related data used to describe a business domain. In order to design a quality database it is crucial to first design a well defined model that captures the Universe of Discourse (UoD). The purpose of modeling languages is to have a language especially suited for *modeling*, that assist the experts in the process of modeling a UoD.

2.2.2 Object Role Modeling

Object Role Modeling (ORM) is a fact-oriented modeling language. It is a further development of NIAM (Natural Language Information Analysis Method). ORM began as a semantic modeling approach which described the world in elementary facts consistent of objects and their relationships to one another. Today it is used mostly for designing databases. A great advantage with ORM is that it has a technique (i.e. method) for translating natural language statements into models and a relational mapping procedure for mapping the models into database schemas.

Fact Types and Bridges

The basic building blocks in ORM are *fact types* and *bridges*. A fact type consists of n concepts with a n-ary relationship between them. A bridge relates a concept to a value type. The most common constructions of fact types are binary, ternary and quaternary fact types. Figure 2.13 shows a binary fact type to the left and a bridge to the right. The concepts are shown as named solid ellipses and value types are shown as a named, dashed ellipses. The graphical representation of elements in ORM is different depending on which generation of ORM that is used. For this chapter and all the following, the graphical representation of the first generation of ORM will be used.

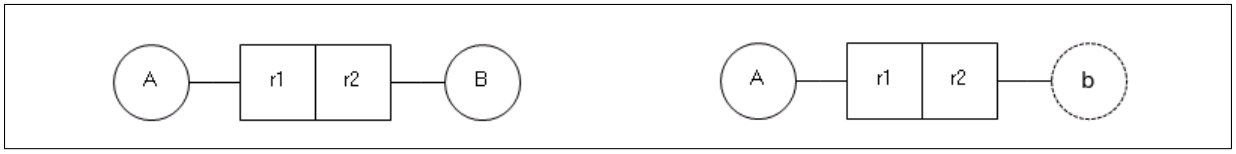


Figure 2.13: A binary fact type to the left and a bridge to the right.

Relationships

A fact type consists of n roles, one for each concept in the relationship. It is common to say that the concept ″plays″ the role it has in a relationship to another concept. As seen on the left in

Figure 2.13 there is a binary relationship between concept `A` and concept `B`, created by the roles `r1` and `r2`. Concept `A` plays the role `r1` in the relationship to concept `B` and concept `B` plays the role `r2` in relationship to concept `A`. A relationship is neutral considering which direction the relationship has, meaning there is no right or wrong way of reading a relationship.

Roles

A role is shown as a named role-box, as seen with role `hasName` and `nameFor` in Figure 2.14. Roles connect a concept to another concept or a value type, and every role is connected to a concept. Figure 2.14 shows the concept `Person` in a relationship with the concept `Name`. The role closest to `Person` called `hasName` is played by `Person` and the role closest to `Name` called `nameFor` is played by `Name`. In pseudo natural language the relationship expresses that ”a Person hasName a Name and a Name is nameFor a Person”.

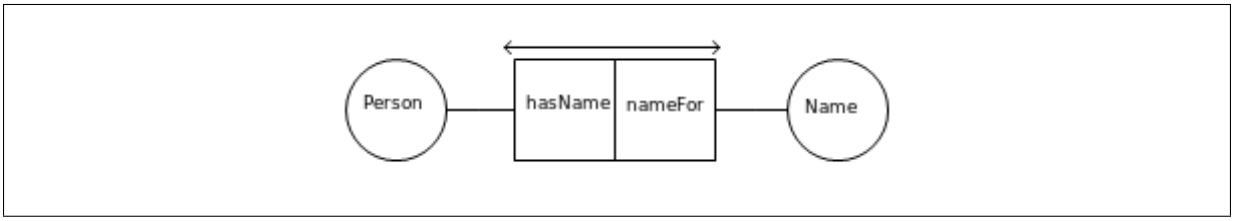


Figure 2.14: The concept Person in a relationship with concept Name.

Constraints

A constraint is a part of the ORM language that helps restrict the set of statements about the Universe of Discourse.

Mandatory Role

The mandatory role constraint is shown by a $\vee$ on the role it restricts. This means that every instance of the concept is obligated to play the role. If there are no mandatory role constraints on a role it means that it is optional for the concept to play it. Figure 2.15 shows an example of a mandatory role constraint on the role `hasName` connected to the concept `Person`. In pseudo natural language it expresses that ”every Person hasName *at least one* Name”.

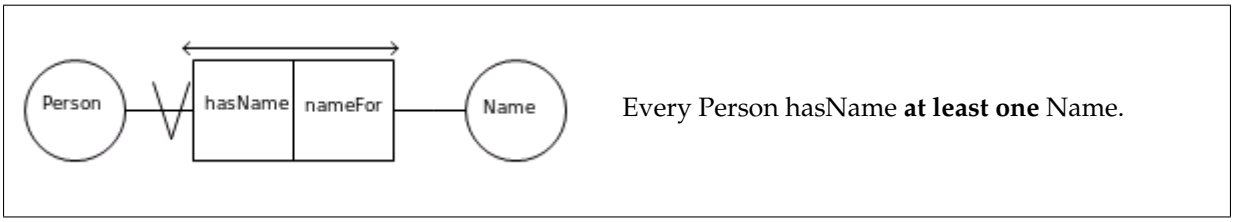


Figure 2.15: An ORM model with a mandatory role.

Internal Uniqueness

An internal uniqueness constraint, shown as ↔ over role-boxes, expresses which role or combination of roles that are required to have unique occurrences. An internal uniqueness constraint on a role means that an instance of a concept can play the role *at most* one time. Figure 2.16 shows the concepts `Person` and `LastName` with an internal uniqueness constraint above the role-box named `hasLastName`. The constraint indicates that an instance of `Person` can only have one `LastName`, since the instance of `Person` can not be repeated in that relationship. In terms of a database this means a Person can only have one LastName, but the LastName can be repeated for other people. If the arrow is above both the role-box named `hasLastName` and `isLastNameFor`, as seen in Figure 2.17, it means that the combination of the two roles can not be repeated for that relationship, resulting in the interpretation that a `Person` can have several `LastNames`, as long as the same `LastName` is not repeated twice. In addition a `LastName` can be applied for several people. To express that one instance of `Person` can only have one instance of `LastName` and vice versa, there need to be one short arrow over the role `hasLastName` and one short arrow over `isLastNameFor`, as seen in Figure 2.18. Now neither the instance of `Person` nor the instance of `LastName` can be repeated in these roles, meaning a `Person` can only have one `LastName` and a `LastName` applies for only one `Person`.

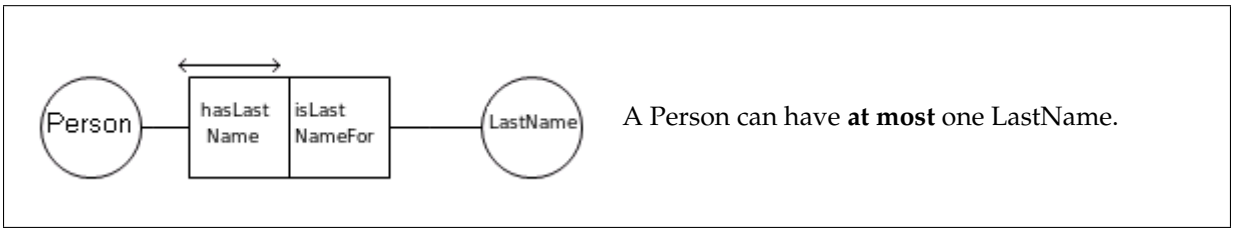


Figure 2.16: An ORM model with an internal uniqueness on one role.

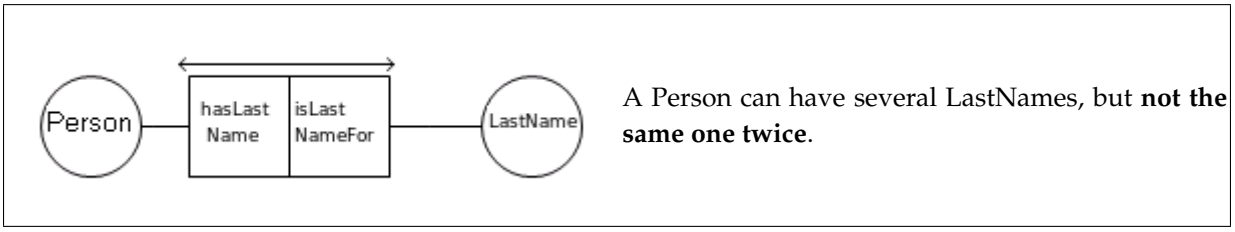


Figure 2.17: An ORM model with an internal uniqueness spanning two roles.

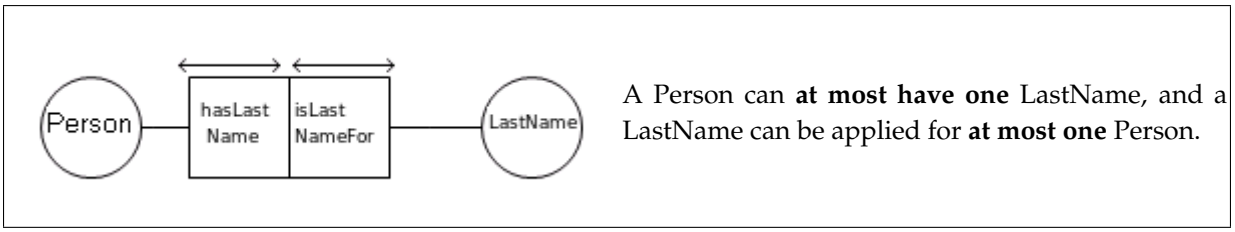
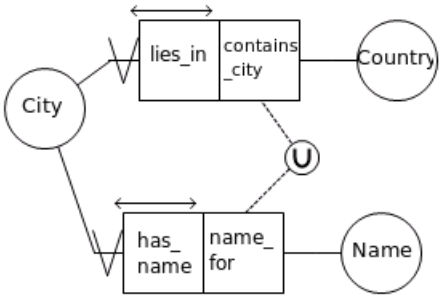


Figure 2.18: An ORM model with an internal uniqueness on both roles.

External Uniqueness

The external uniqueness is shown as a circled U on a dotted line that connects two or more roles. The constraint expresses that a combination of roles, which is not necessarily played by

the same concept, is required to be unique. In Figure 2.19 there is an example of the concept `City` that is made unique by the `Country` it lies in and the `Name` it has. The combination of the instances in `lies_in` and `has_name` can occur only once for a `City`.

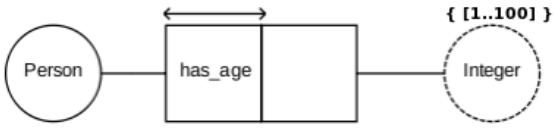


A City **uniquely determined** by the **Country** it lies in and the **Name** it has.

Figure 2.19: An ORM model with an external uniqueness.

Value Constraint

A value constraint indicates which values are allowed in a value type or role. This constraint should be used only if the value list is reasonably stable, to avoid changing the constraint too much. The set of possible values is declared as a list enclosed by curly brackets {} or as a range enclosed with square brackets [] inside a pair of curly brackets. The square brackets are used when the values are real numbers. Figure 2.20 shows an example of a value constraint on the value type `Integer`.



A Person's age must be **between the number 1 and 100**.

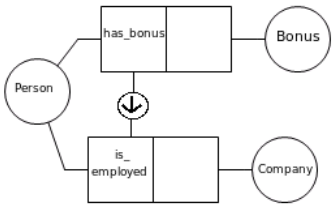
Figure 2.20: An ORM model with a value constraint.

Set-Comparison Constraints

Set-comparison constraints restricts the way the population of one role, or role sequences, relates to the population of another.

Subset Constraint

A subset constraint is visualized by a circled directed arrow going from one role to another role played by the same concept. The constraint means that all instances that play the role at the tail of the arrow, must also play the role at the head of the arrow. Figure 2.21 shows that the concept `Person` can play the role `has_bonus` only if the person also plays the role `is_employed`. A `Person` is allowed to play the role `is_employed` without playing the role `has_bonus`.

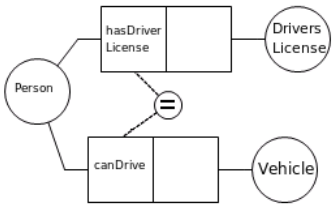


A Person that hasBonus **must also** be isEmployed.

Figure 2.21: An ORM model with a subset constraint.

Equality Constraint

The equality constraint is visualized by a circled equality sign (=) on a dotted line that connects two or more roles played by the same concept. This constraint expresses that the population that plays these two roles must always be equal. If an instance play one of the roles it must also play the other. In Figure 2.22 there is an example of a `Person` that has a `drivers license` and `can drive`. This structure expresses that every person that can drive a vehicle has a driver's license and every person that has a driver's license can drive.

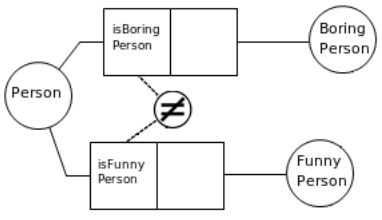


Every Person that canDrive **also** hasDriversLicense, and vice versa.

Figure 2.22: An ORM model with an equality constraint.

Exclusion Constraint

An exclusion constraint is visualized by a circled equality sign with a line across it, on a dotted line that connects two or more roles played by the same concept. This constraint expresses that an instance is not allowed to play more than *at most* one of the roles touched by this constraint. Figure 2.23 shows that a `Person` may play the role `isFunnyPerson` or the role `isBoringPerson`, but not both. The roles are not mandatory so a person may play none of them.

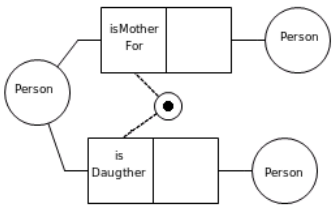


Every Person may be isFunnyPerson **or** isBoring-Person, **but not both**.

Figure 2.23: An ORM model with an exclusion constraint.

Inclusive-or Constraint

An inclusive-or constraint, or disjunctive mandatory role constraint, is visualized as a circled dot on a dotted line, that connects two or more roles played by the same concept. This constraint expresses that an instance of a concept *must* play *at least* one of the roles, or several of the roles, touched by this constraint. Figure 2.24 shows that a `Person` must be a mother for someone, or a daughter to someone, or both.

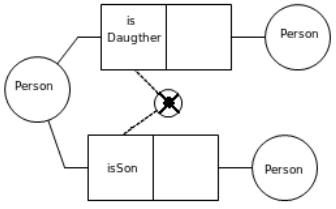


Every Person **must be** isMotherFor **or** isDaughter, **or both**.

Figure 2.24: An ORM model with an inclusive-or constraint.

Exclusive-or Constraint

An exclusive-or constraint is a combination of an inclusive-or constraint and an exclusion constraint. It is visualized as a circled dot with a "X" symbol. This constraint expresses that an instance of a class plays *exactly* one of the roles touched by this constraint. An instance must play a role, and no more than one. Figure 2.25 shows that a `Person` must be a daughter to someone or a son to someone. The person can not be neither or both.



Every Person **must** be isSonTo **or** isDaughterTo, **but not both**.

Figure 2.25: An ORM model with an exclusive-or constraint.

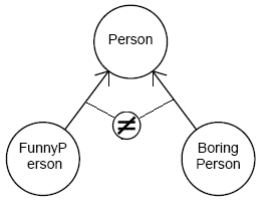
Subtyping Constraint

A concept is a subtype when it is classified into a more specific type. There are three types of subtype constraints: exclusive subtypes, exhaustive subtypes and partition.

Exclusive Subtypes

An exclusive subtype is visualized the same way as an exclusion constraint, only that the dotted line connects two or more concepts. Subtypes are exclusive when an instances of the supertype must be a member of *at most* one of the subtypes. An instance can be a member of neither subtypes. Figure 2.26 shows that all members of the concept `Person` may be a `FunnyPerson`

or a `BoringPerson`, but not both. There may be instances of `Person` that are neither funny nor boring.

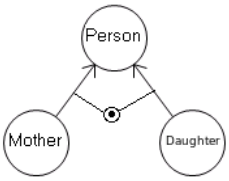


Every Person **may be** a FunnyPerson **or** a BoringPerson, **but not both**.

Figure 2.26: An ORM model with an exclusive subtypes.

Exhaustive Subtypes

An exhaustive subtype is visualized the same way as the inclusive-or constraint only that the dotted line connects two or more concepts. Subtypes are exhaustive when all instances of the supertype *must* be in *at least* one of the subtypes. The union of the subtypes should be equal to the supertype. An instance can be a member of all the subtypes, but it can not be a member of none. Figure 2.27 shows the exhaustive constraint between the subtype `Mother` and `Daughter` which expresses that every person must be either a `Mother` or a `Daughter` or both.

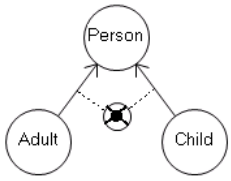


Every Person **must be** a Mother **or** a Daughter **or both**.

Figure 2.27: An ORM model with an exhaustive subtypes.

Partition Constraint

A partition constraint is visualized the same way as the exclusive constraint only that the dotted line connects two or more concepts. Subtypes are partition when all instances of the supertype have to be a member of *exactly* one of the subtypes. This constraint is a combination of exclusive subtypes and exhaustive subtypes. In Figure 2.28 there is an example of a partition constraint between the concepts `Adult` and `Child` that expresses that every `Person` must be either an `Adult` or a `Child`, but not both.



Every Person **must be** either an Adult **or** a Child, **but not both**.

Figure 2.28: An ORM model with a partition constraint.

Other Constraints

Frequency Constraint

A frequency constraint, also known as occurrence frequency constraint or cardinality constraint, expresses that an instance of a concept must play a certain role n times. If $n = 1$ it is equivalent to an internal uniqueness constraint. It is possible to express that an instance must play a role at most n times or at least n times. When they are combined they express a frequency range. Figure 2.29 shows an example of a `Bus` that has a frequency constraint of at most 50. It means that for every bus that has passengers, the number can not exceed 50.

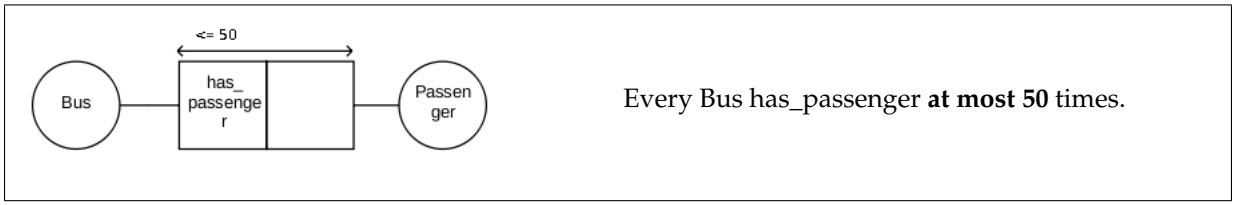


Figure 2.29: An ORM model with a frequency constraint.

Equivalence-Of-Path

Equivalence-Of-Path (EOP), a special case of join equality constraint, is often used when modeling reservation, logging and ticketing. It is relevant when resources are being reused in a model, and there exist more than one path from one concept through several binary fact types with one-to-many constraints to another concept. The constraint expresses that the different paths should lead to the same instance in the common concept. There exist no standard way of visualizing this constraint, so in Figure 2.30 it is visualized with a red line with arrows indicating the direction of the path. The figure shows an equivalence-of-path on two paths from `Ticket` to `Cinema`. The first path goes from `Ticket` through the role `applies_for` to `Show` and from `Show` to `Cinema` by the role `shown_in`. The second path goes from `Ticket` to `Seat` through the role `applies_for` and from `Seat` to `Row` through the role `part_of`. Finally the path goes from `Row` to `Cinema` through the role `situated_in`.

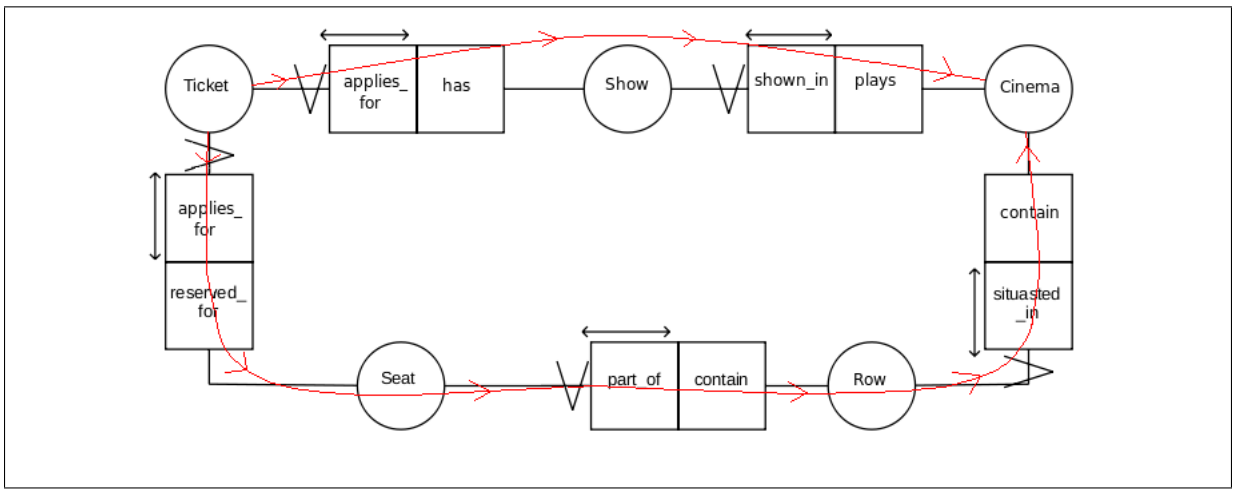
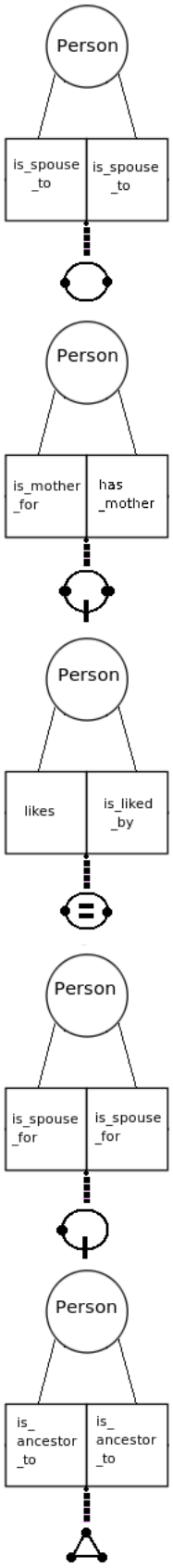


Figure 2.30: An ORM model with an equivalence-of-path constraint.



Symmetric: If `Person` A is *spouse* to another `Person` B, the other `Person` B is also a spouse to the first `Person` A.

Asymmetric: If `Person` A is *mother* to another `Person` B, the other `Person` B is **not** mother to the first `Person` A.

Reflexive: Every `Person` *likes* him/herself.

Irreflexive: `Person` A can **not** be his/herself *spouse*.

Transitive: If `Person` A is *ancestor* to another `Person` B, and `Person` C is ancestor to the `Person` B, then `Person` A is also ancestor to `Person` C.

Figure 2.31: An ORM models with ring constraints.

Ring Constraints

When two roles are played by instances of the same concept, the path from the concept through the role pair and back to the concept forms a "ring". See Figure 2.31 for examples on ring constraints.

Symmetric. R is symmetric iff for all x,y: $xRy \rightarrow yRx$.

Asymmetric. R is asymmetric iff for all x, y: $xRy \rightarrow \sim yRx$.

Reflexive. R is reflexive iff for all x: xRx.

Irreflexive. R is irreflexive iff for all x: $\sim xRx$.

Transitive. R is transitive iff for all x, y, z: xRy and $yRz \rightarrow xRz$.

2.3 OWL vs. ORM

Even though OWL originates from the field of Artificial Intelligence and ORM originates from the field of databases they are both languages for modeling information. Their difference in origin leads to some fundamental differences such as Open World Assumption in OWL and Closed World Assumption in ORM. This section gives a presentation over some of the differences between them. For an overview of the differences between the languages, see Table 2.1.

2.3.1 Open World Assumption vs. Closed World Assumption

OWL, as the ontology language for the Semantic Web, uses the Open World Assumption (OWA). OWA makes the assumption that if knowledge is not found, it does not mean that it is false, it is merely *unknown* (Hitzler et al., 2009). The idea with the Semantic Web is that people will add information bit by bit, so one can assume that not all information is available. Therefore information is stated as unknown and not false, since it might be added later. ORM on the other hand, works with the Closed World Assumption (CWA). CWA is the assumption that a knowledge base or database is *true* concerning all the data in the database (Hitzler et al., 2009), so data that is not in the database is considered to be false. This allows one to state negative claims. In the CWA everything which is not explicitly stated to be true, is considered to be false (Hitzler et al., 2009).

2.3.2 Non Unique Name Assumption vs. Unique Name Assumption

In addition to the OWA, OWL has a Non Unique Name Assumption (NUNA). NUNA means that unless explicitly stated, one does not know that two different names refer to two different "things" (Hebeler et al., 2009). Names are like pointers in programming languages, which all can refer to the same, even though the names are different. This assumption makes it necessary to state that things are the same or different from one another. NUNA makes it easier to handle

ambiguous or redundant data, which according to Hebeler et al. in (Hebeler et al., 2009) is common in information systems. It is possible to create resources that are the same without destroying any information. ORM on the other hand, uses a Unique Name Assumption (UNA). UNA assumes that two concepts with different names are different and two concepts with the same name are the same. In ORM all concepts that are visually separated and in addition have different names are implicitly different, with the exception of subtypes. It is not necessary to state this in any way.

2.3.3 Properties vs. Roles

In OWL properties exist independent of the classes, which means that they can apply for none or several classes. The idea behind the independence is that one can state in the model that there is a relationship between individuals before one knows which class the individual belongs to. This independence allows the properties to have inheritance of their own. A relationship in OWL is directed so it only describes one way, from domain to range. Roles in ORM, on the other hand, are always connected to a concept. They are used to link concepts together, to create relationships between them. A relationship is neutral when it comes to the direction of the relationship.

2.3.4 Reasoning

OWL has reasoning which performs consistency checks on the semantic in the model in addition to inferring and adding information that is only implicitly stated. Tools that model ORM have consistency checks on the syntax, but not on the semantic. There are no tools for ORM that support reasoning which can infer new knowledge. All information in an ORM model needs to be stated explicitly in order to be known.

2.3.5 Database Support

There exist tools that can map from a database to OWL, so that OWL can extract and use the data from a database. ORM on the other hand has an algorithm or mapping procedure that allows it to convert the model to a database schema.

2.3.6 Hierarchy and Inheritance

OWL has hierarchies in both classes and properties which enables specialization and inheritance. A class can be a subclass of another class and inherit all of the properties and restrictions of the superclass. A property can be a subproperty of another property and inherit the domain and the range from the superproperty. All of the inheritance is top-down which means a superclass or a superproperty can not inherit from its subclass or subproperty. ORM does not have hierarchy in roles, but they have hierarchy in concepts. It is possible to have subtypes, meaning a concept can be a subtype of another concept. The subtype will inherit the roles of the supertype. In addition ORM has subsets that are similar to the hierarchy in properties. Subset creates sets of instances that are subset of other supersets.

	Reasoning	**Database Support**	**World Assumption**	**Name Assumption**
OWL	Yes	Mapping	Open World Assumption	Non Unique Name Assumption
ORM	No	Algorithm/ Mapping Procedure	Closed World Assumption	Unique Name Assumption

Table 2.1: The differences between OWL and ORM.

Chapter 3

Problem Description and Requirements

As seen in Chapter 2 there are some fundamental differences between the languages OWL and ORM. OWL works with an Open World Assumption (OWA) and the Non Unique Name Assumption (NUNA) while ORM works with a Closed World Assumption (CWA) and the Unique Name Assumption (UNA). It turns out that these differences have an impact on the languages' ability to create information models and support information systems. This chapter will give an introduction to the problems considered in this thesis, followed by an overview over the different requirements for the modeling tools and mapping process.

3.1 Problem

The goal of this thesis is, as mentioned in Chapter 1, to compare the languages ORM and OWL. This comparison can be divided into two subgoals. The first subgoal is to compare the languages' capability and suitability to model information. This will be investigated in two ways: first by mapping the general structures and constraints from ORM to OWL, and then by mapping a specific example. This example is created by capturing some of the rules of the Norwegian National Register. The ORM and OWL models will be compared to check if there are any structures or restrictions in ORM that can not be expressed in OWL. In order for OWL to be successful as an information modeling language the resulting OWL model should contain all the business rules captured in the ORM model. How suitable OWL is also depends on how it can visualize the same structures and constraints, and how different the constructions are in the two languages. It is reasonable to assume that if the constructions in OWL are significantly different from the ones in ORM that OWL may not be very suitable for the task, even though it is capable. If it is necessary to do a lot of tweaking and reconstruction in order to express a certain statement in OWL, then the essence of the statement may not be very visible in the model. This leads to poor readability and understanding of the domain. In addition this might make it difficult to model directly in OWL.

There are to our knowledge no available tools that are capable of doing this mapping automatically. The only tool that can automatically map from ORM to OWL is an unavailable tool called DogmaModeler (Jarrar, 2007, 2005; Hodrob & Jarrar, 2010). The mapping is therefore performed manually by following predetermined mapping rules that state what a structure in ORM is translated to in OWL.

The second subgoal is to compare how the languages can support information systems. By support we mean the ability to provide semantic information about the domain. Semantics is provided when the model is combined with the data. It is advantageous to have both the semantics and the data when distributing, since the relationship between the data is then independent of the storage, and allows the data to automatically be incorporated into another system.

ORM is used to automatically create database schemas. It is capable of mapping the structures in the model to tables in a database schema, but after the schema is created the connection is lost. To our knowledge there are no tools that allow ORM models to connect to the data in the database after the database is created. However, there may be aspects of OWL that are more flexible in this area. This will be investigated by creating a database from the ORM model of the Norwegian National Register, and attempt to connect the data in it to the OWL model mapped from ORM.

OWL models are in contrast to ORM models, more dynamic, more "alive". After the creation of an OWL model the data can be added to the model itself. New connections can be added directly in the model and immediately effect data, since the data is a part of the model. The question is how well this dynamic management of data works if the data resides in a database. An OWL model can be connected to a database by mapping. There are two main ways of performing the mapping. One approach extracts the data from the database and creates a populated ontology with the data. The other approach lets the data reside in the database and creates a bridge to it. The advantage of creating a populated model is that it will remove the need for a separate database. The data will be re-stored in a new data structure and the need for maintenance of the possibly old database is gone. On the other hand, databases can store huge amounts of data, possibly more than an ontology, and can perform better on (complex) queries than most tools for ontologies. The disadvantages of not extracting the data is that there are at present time no tools that allows the structure of the database to be changed from the ontology, which means that when the tables are added or removed from the database the mapping must be recreated. Although this happens rarely, it is fairly time consuming. No matter which method is used for the mapping, it allows the data in the database to be easier shared across the Web, the data can be reasoned over and as an OWL model it can be combined with other models.

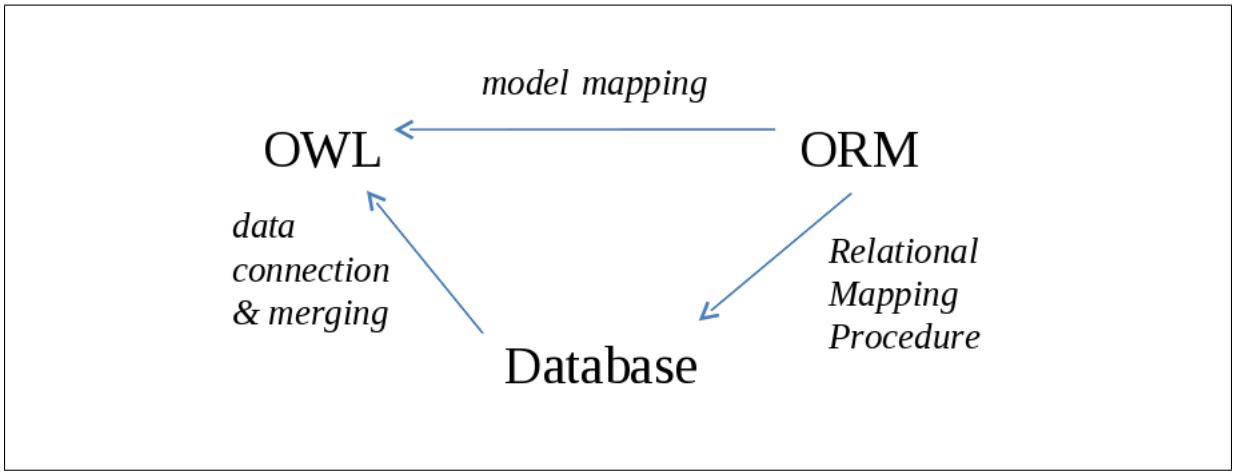


Figure 3.1: The mapping processes, ORM to OWL, ORM to database and database to OWL.

Figure 3.1 shows the mapping processes performed in order to investigate the problems. An ORM model inspired by the Norwegian National Register will be created. Then the ORM

model will be mapped to an OWL model to test OWL's abilities to model information. This is represented by the arrow from *ORM* to *OWL* called *model mapping*. Further the ORM model will create a database by using the Relation Mapping Procedure, which is represented by the arrow from *ORM* to *Database* called *Relational Mapping Procedure*. Last, we will attempt to connect the database to the OWL model to see if the data can be merged with the model, which is represented by the arrow from *Database* to *ORM* called *data connection & merging*.

3.2 Modeling Tool Requirements

In order to create models in the languages ORM and OWL it is necessary to find some tools that support modeling in the different languages. This section begins with an introduction of the requirements for modeling tools for ORM, followed by the requirements for modeling tool for OWL.

Object Role Modeling

The requirements for a modeling tool for ORM is that the tool includes as much of the language as possible. In addition it is an advantage if the tool is free of charge and has implemented the algorithm for creating databases based on the model.

The Web Ontology Language

The requirements for a modeling tool for OWL, is that the tool allows the writing of the second generation of OWL, OWL 2. OWL 2 contains more properties and constraints than the first generation. In addition the tool should support writing in Manchester and Turtle syntax, since Manchester is a fairly easy syntax to write restrictions in, while Turtle is very easy to read and understand. The tool needs to be orderly and easy to navigate, and one should be able to easily see the connection between the different classes, properties and individuals. The output file should also be clear and orderly and not proprietary. Last, but not least, the tool should be free of charge and well documented.

3.3 Mapping Requirements

There are some requirements for the outcome of the mappings, both for the mapping from ORM to OWL and for the mapping from database to OWL. First is a presentation of the requirements for the mapping between ORM and OWL, followed by the requirements for the mapping between a database and OWL.

ORM to OWL

The requirement for the mapping between ORM and OWL is that the mapping is correct and feasible. That is, the resulting OWL model should contain equal constructions to the most vital constructions in the ORM model. In addition the OWL model should be able to express all of the statements in ORM with either an equal construction or with other constructions that produce the same statement. No information should ideally be lost in the mapping process.

Database to OWL

A requirement for the mapping from a database to OWL is that all of the data in the database should either be available in the OWL model or accessible through the OWL model. No data should be lost in the mapping process. In addition it should be possible to query the resulting OWL model for all the data in the database within the limits of the SPARQL query language. We restrict ourselves to relational databases since it is the kind of databases the Relational Mapping Procedure in ORM creates.

Chapter 4

Tools

We started with a literature survey of current modeling and mapping tools, and came across some well known and used tools, some promising ones and some interesting ones. On the basis of this survey we chose to have a closer look at the tools listed below. First is a selection of the tools relevant for the modeling process. Second is an overview of the tools for mapping from ORM to OWL. Last there is a selection of some tools for connection and mapping from databases to OWL models.

4.1 Tools for Modeling ORM

For modeling ORM the tools that have been viewed and read about are: stORM, NORMA, ORM Lite and Edraw Max.

4.1.1 stORM

stORM is a development tool created by It Liberator (IT Liberator, 2011) that supports all phases of a system development. It is a CASE tool that provides a graphical representation of data and it allows modeling in the first generation of ORM. Its output is proprietary and it has a feature that can generate SQL schemas. Further stORM supports binary fact types, and the simplest cases of equivalence-of-path constructions, but it does not support subtyping constraints or ring constraints.

4.1.2 NORMA

NORMA is a plug-in for Microsoft Visual Studio developed by the ORM Foundation (NORMA, 2011). It is an open source tool with a diagram editor that supports the notation used in the second generation of ORM (NORMA -The Software!, 2011; NORMA, 2011). Further it includes the mapping ability from the ORM model to four different relational database management systems (RDBMS), namely MS SQL Server 2005, DB2, Oracle, and PostgresSQL (NORMA, 2011). Although the plug-in is free of charge, Visual Studios is not, so this tool requires a license

for Microsoft Visual Studio. NORMA supports unary, binary and ternary fact types, subtyping constraints and ring constraints. It is also possible to make objectifications in this tool.

4.1.3 ORM Lite

ORM Lite is a light-weight ORM modeling tool that aims to popularize ORM (ORM Lite, 2011). It is meant to give an introduction for new ORM modelers and especially students. It supports most of the ORM 2 notation, and includes an ORM verbalizer, a Relation Table Diagram and a diagram that shows the implications of constraints on a database. It is open source and cross-platform, meaning it works on several platforms such as Mac, Linux, Unix and Windows. In addition it is scriptable. The tool can be downloaded free of charge from the ORM Foundation's website.

4.1.4 Edraw Max

Edraw Max is a modeling and design tool developed by Edraw Soft (Edraw Max, 2011). It is capable of creating ORM Diagrams and design databases. It supports binary, ternary and quaternary fact types, ring constraints, set-comparison constraints, frequency constraints and subtyping constraints. It is possible to download a trial version of the tool free of charge or to purchase the entire version from Edraw Soft's website.

4.2 Tools for Modeling OWL

The OWL modeling tools that we have reviewed are: Protege, OntoStudio, NeOn ToolKit and TopBraid Composer.

4.2.1 Protégé

Protégé was developed by Stanford Center for Biomedical Informatics Research at the Stanford University School of Medicine (Protege, 2011). Its version 4.0 and beyond allows use of the second generation of OWL. Version 4.1 supports the ability to save the model in several formats, among others, XML/OWL, XML/RDF, Turtle and Manchester syntax. This is very handy as Turtle is the easiest format for humans to read, while XML might be needed for distributing data (Protege, 2011; Protégé OWL Tutorial, 2011). The output from this tool in not proprietary and can be viewed in standard editors. The tool is free of charge and available for download from Protégés website.

4.2.2 OntoStudio

OntoStudio is a commercial modeling environment for creating and maintaining ontologies (OntoStudio, 2011) developed by Ontoprise. The tool contains, among other functions, a mapping tool, a graphic rule editor and import functions for schemas, models and structures.

The editor supports OWL, RDF(S), RIF and ObjectLogic. In addition the tool allows plug-ins. From Ontoprise's website it is possible to purchase the tool or download a three months evaluation version free of charge.

4.2.3 SemanticWorks 2011

SemanticWorks 2011 is a tool for creating and editing Semantic Web ontologies developed by Altova (SemanticWorks 2011, 2011). The tool is graphical and allows the user to create OWL models by using icons, which then automatically generates RDF/XML or N-triples code. The user can edit existing ontologies or create their own from scratch. The tool supports all three dialects of OWL and performs syntax and consistency checks. It is possible to purchase and download the tool from Altova's website.

4.2.4 TopBraid Composer

TopBraid Composer is a commercial graphical development environment for Semantic Web ontologies (TopBraid Composer, 2011) developed by TopQuadrant. The tool is a RDF and OWL ontology editor that follows the W3C standards and is implemented as an Eclipse plug-in. Composer is a part of the TopBraid Suite that means it incorporates an extensible framework for developing semantic client/server or browser solutions that can integrate separate applications and data sources. TopBraid Composer is available for trial and purchase from TopQuadrant's website.

4.3 Tools for Mapping from ORM and OWL

A research was conducted to find a tool that map from an ORM model to an OWL model. A criterion for the tool is that it can map all the structures and restrictions in the ORM model to the OWL model. The tool needs to be free to download, available for all and made in English. In addition it needs to be user friendly and the output should be easy to get an overview of. To our knowledge there are no tool that satisfy all the requirements, the closest one is DogmaModeler.

4.3.1 DogmaModeler

DogmaModeler is a Java modeling tool developed by Mustafa Jarrar (Jarrar, 2007, 2005; Hodrob & Jarrar, 2010). This tool represent ontologies and can reason over them (Hodrob & Jarrar, 2010). In addition it map from ORM to OWL 2 by following a set of mapping rules described in (Hodrob & Jarrar, 2010).

DogmaModeler allows the user to create ORM diagrams. The tool can translate the diagram into pseudo natural language, automatically map from ORM to OWL 2 DL, and reason over the OWL 2 DL by using a reasoner called RacerPro2. The mapping uses the ORM markup language which it extracts from the graphical notation of the ORM diagram. Hodrob and Jarrar mentions in (Hodrob & Jarrar, 2010) that it is possible to create queries using RacerPro Query Language.

This tool is the closest one to our knowledge that is able to map directly from ORM to OWL. Unfortunately it seem like this tool is developed with the purpose of being used in a PhD and therefore not possible to run outside of the Vrije University Brussels in Belgium. In addition the tool is poorly documented although the mapping rules which it uses are described in (Hodrob & Jarrar, 2010).

4.4 Tools for Mapping and Connecting to Databases

We researched tools that can map from a database to an OWL model. The criterion is that it must be able to map from relational databases to OWL, without loosing information (i.e. data). That is the resulting OWL model should contain all the information that is in the database. Other criteria are that the tool is free of charge, available for all, made in English, and well documented. In addition it is desired to look at different ways of dealing with the data in the database. Two approaches were discovered. One approach extracts the data from the database and creates a populated ontology, while the other approach keeps the data in the database and creates a bridge to it. Following is a selection of the tools we found during our research; some of them met more of the requirements then others.

4.4.1 QuOnto

QuOnto, or Querying Ontologies, is an ontology representation and reasoning tool prototype developed at the University of Rome (QuOnto, 2011). The tool accesses database management systems (DBMS) by using a fragment of DL-Lite called $DL - Lite_A^+$. It is wrapped by a server called DIG-QuOnto which has a Java-based DL reasoner for DL-Lite that can access external data sources (Calvanese et al., 2009).

Further the tool supports the second generation of OWL's profile QL and implements a query rewriting algorithm. The algorithm is capable of both consistency checking and querying over DL-Lite knowledge bases (QuOnto, 2011). The user is offered several different user interfaces such as, a native XML parser and serializer, an OWL/SPARQL interface and a DIG interface (Quonto QUerying ONTOlogies, 2011).

QuOnto supports a specific query language SparSQL (SPARQL + SQL) that allows it to express full SQL-like queries over the ontologies, while preserving the efficiency of the query answering process (Quonto QUerying ONTOlogies, 2011). The query algorithm takes as input a DL-Lite knowledge base $Knowledgebase(K) = (T - box, A - box)$ and a union of conjunctive queries q, and returns the answers, i.e., the set of tuples that satisfy the query in every model of K. The data remains in the database and the tool creates a bridge to it.

QuOnto can be retrieved for testing purposes by directly contacting the authors.

4.4.2 D2RQ

D2RQ is a tool mentioned in the survey of current approaches for mapping of relational databases to RDF, done by W3C, from 2009 (Sahoo et al., 2009). It is developed by Technische

Universität Berlin and Freie Universität Berlin and can be downloaded free of charge from Sourceforge's website. The tool maps from a database schema to a RDF vocabulary or an OWL model. The result of the automatic mapping is a RDF graph. The platform of D2RQ consists of three parts, D2RQ Mapping language, D2RQ Engine and the D2R Server (Bizer, Cyganiak, Garbers, Maresch, & Becker, 2011).

The mapping language is a declarative mapping language for describing the relation between ontologies and relational data models. The engine is a plug-in for Jena and Sesame Semantic Web toolkit, that rewrites Jena and Sesame API calls to SQL queries, execute the queries against the database and returns the results up to the higher layers of the framework. The engine creates a mapping file that describes the mapping from a relational database to either a RDF vocabulary or an OWL model. The server is a HTTP that can be used to provide the user with a Linked Data View, a HTML view and a SPARQL Protocol Endpoint over the database (Bizer et al., 2011; Cyganiak & Bizer, 2006).

The D2RQ mapping language supports conditional mappings, mapping of multiple columns to the same property and mapping of table structures below the first normal form (Cyganiak & Bizer, 2006). It uses translation tables in the mapping process (Cyganiak & Bizer, 2006).

The part of the tool that is interesting in this case is the engine, since it rewrites queries and creates a mapping file from a relational database to an OWL model. In addition to rewriting Jena and Sesame API calls, the engine is capable of rewriting SPARQL queries into SQL queries which is executed against the database. The SPARQL queries are processed by Jena's ARQ query engine (Bizer et al., 2011). The mapping file does not read the data from the database into the OWL model, it only creates a bridge to it.

The mapping file is a RDF document that consists of one or more `ClassMaps` and one or more `PropertyBridges` (Bizer et al., 2011). A ClassMap specifies how instances of a class are identified (Bizer et al., 2011). It also represents a mapping from a set of entities described within the database, to an OWL class (Bizer et al., 2011). There is usually one ClassMap for each table in the database (Hebeler et al., 2009; Cyganiak & Bizer, 2006). Each ClassMap has a set of PropertyBridges, which specifies how the properties are created and how they are connected to the table columns in the database (Bizer et al., 2011). PropertyBridge values can be created directly from database values or by employing patterns or translation tables (Bizer & Cyganiak, 2006). Further the mapping generates class names from the table names and property names from column names (Bizer & Cyganiak, 2006). Every mapping file starts with listing the prefixes used in the ontology.

The drawbacks when generating RDF from database schema is that the resulting RDF graph is an almost exact reflection of the structure in the database (Hebeler et al., 2009). Other disadvantages are that it is not possible to connect multiple databases to one model (Bizer & Cyganiak, 2007) and changes to the database structure requires a new mapping in order to be captured in the model. In addition the engine that rewrite queries from SPARQL to SQL have some limitations that lead to poor performance on complex queries with `OPTIONAL`, `FILTER` and `LIMIT` (Bizer & Cyganiak, 2007). In order to build useful queries it is required to have a close awareness of the structure of the database (Hebeler et al., 2009).

4.4.3 RDBToOnto

RDBToOnto is another tool mentioned in the survey of current approaches for mapping of relational databases to RDF, done by W3C, from 2009 (Sahoo et al., 2009). It creates populated ontologies in RDF/OWL from a relation database and is developed by Farid Cerbah as a part of the TAO (Transitioning Applications to Ontologies) project (RDBToOnto, 2011). It is freely available for download at the TAO project's website (RDBToOnto, 2011).

The tool is implemented in Java and uses patterns to attempt to categorize the data into new structures (RDBToOnto, 2011). Cerbah says in (Cerbah, 2008) that the main motivation behind the tool is to implement a process which populates ontologies by both exploiting the database schema and the data in the database when identifying the ontology structure.

This is done by an interactive approach that allows the user to adjust the learning process by defining constraints that help the tool to categorize by patterns it have missed. The user can add local constraints which specify how names should be derived from the attribute values. The learning method is a data-driven mechanism which automatically creates categorization patterns from the database content. The key is to identify the relation attributes hidden in the data, which may be a good categorization sources.

RDBToOnto supports the whole process from accessing the data in the database to generating populated ontologies with the data from the database (Cerbah, 2008). By providing the tool with the URI of the database as input, and with the default settings, a user can get a populated ontology.

The resulting ontology is created in two parts. First the structures get mapped from the database schema to the ontology. The tables defined in the schema are mapped to classes, the relation attributes is mapped to datatype properties and the binary associations between tables are turned into object properties between classes. Then some of the classes are modified to create a class hierarchy.

The database goes through a normalization step in order to eliminate redundancy and duplicated data in the source tables. As a result some object properties are created where there otherwise would be duplication of data (Cerbah, 2008). The duplication mentioned here is the duplication of data within the ontology.

Cerbah (Cerbah, 2008) states that a mapping from a relational database to an ontology will result in an ontology with a structure quite similar to the structure of the relational database. The resulting ontology therefore does not have the expressive power or the ability to take the full advantage of the RDF/OWL formalism.

4.4.4 RDB2ONTO

RDB2Onto is a tool that creates semantic metadata from the data in relational databases. It was created as a part of the NAZOU project and is available for download at the project's websites (RDB2Onto, 2011).

This tool converts data from the database to a RDF/OWL ontology document based on a predefined template (Šeleng, Laclavík, Balogh, & Hluchỳ, 2007). It maps from a database to

a RDF/OWL XML-based template by executing a SQL queries and use the result set as a base for the mapping. The result set is filled into the template by putting a value for a given row into the elements in the template that are enclosed with brackets. The template is then translated, one row at the time, and stored in a XML-based ontology model (Šeleng et al., 2007). The ontology model is populated with the result of the query. The desired effect of the approach is a solution which is, in the opinion of Laclavik in (Laclavik, 2007) simple, in contrast to the complicated mapping of D2R MAP (Laclavik, 2007).

4.4.5 D2OMapper

D2OMapper is a tool created by Xu et al. which automatically and semi-automatic (via man-machine conversation) maps from relational database schema, like ER, to an OWL ontology (Xu, Zhang, & Dong, 2006). It is developed on J2SE version 1.4.2 and the Jena2 API (Xu et al., 2006). D2OMapper takes as input a database schema, an already existing ontology, and optionally a renaming trails file, which tells it how to change some specified names. The automatic mapping is done by following a set of predefined heuristic rules which are based on the conceptual correspondences between the schema and the ontology. The result is a XML document of the mapping. The development of D2OMapper was done in the same project as developed the tool ER2WO which translate from ER schemas of relational databases to OWL ontologies.(Xu et al., 2006)

This tool is poorly documented and it is difficult to find information on the process of mapping, or where and how to get hold of the tool.

4.4.6 VisAVis

VisAVis is a Protégé plug-in that maps from relational databases to OWL ontologies (Konstantinou, Spanos, Chalas, Solidakis, & Mitrou, 2006). The language used is OWL DL and the data can be retrieved from the ontology by using SPARQL. The tool was developed by Konstantinou and his coworkers and it is available at Konstantinou's Website.

The mapping process consists of four steps. The first step captures the data from the database. This can be done by manually select the desired subset of records. In the second step the user selects a class from the ontology to map the records to. The third step consists of consistency checking, along with evaluation, validation and refinement of the mapping. The fourth and last step is to modified the ontology and add additional information.

The schema of a relational database is compared to the T-box of the knowledge base and the instances are compared to the actual data in the A-box. The resulting ontology does not contain an A-box, but a reference to the dataset in the database through the T-box. The reference is a class property assigned a String that contains the actual SQL query that returns the dataset from the database.

Tools	Mapping Languages	Populated Ontologies	Querying Abilities
	RDB to OWL		
QuOnto	NO (DL-Lite KB)	NO	Rewrite with SparSQL
D2RQ	YES	NO	Rewrite SPARQL to SQL
RDBToOnto	YES	YES	SPARQL?
RDB2ONTO	YES	NO	Unknown
D2OMapper	YES	NO	Unknown
VisAVis	YES	NO	SQL query in each class

Table 4.1: Overview of the tools features

4.4.7 Summary and Comparison of the Tools

The tools have several similarities and some differences. The tables 4.1, 4.2 and 4.3 shows a comparison of the tools. To start with the similarities; all of the tools are available in English and all of the tools with exception of QuOnto, which maps to a DL-Lite knowledge base, maps from databases to OWL. Except for D2OMapper all of the tools are freely available and accessible. To our knowledge it is not possible to get hold of D2OMapper. Regarding the differences; QuOnto, D2RQ, RDB2ONTO, D2OMapper and VisAVis all keep the data as an A-box in the database, while RDBToOnto creates RDF/OWL ontologies populated with data from the database.

Querying Ability

QuOnto and D2RQ are the two tools that query by rewriting queries. QuOnto uses a query language called SparSQL which is a combination of SPARQL and SQL, while D2RQ rewrites SPARQL queries to SQL that are executed against the database. RDBToOnto produces populated ontologies so it should be able to query the ontology with SPARQL, although this is not conformed. RDB2ONTO produces a RDF/OWL ontology that have the A-box in the database, and whether or not it rewrites queries is unknown. D2OMapper does not mention querying. VisAVis has a SQL query as a value in a property of each class so it is assumed that it queries the ontology with SPARQL, although this is not conformed, and use the SQL query to get the result from the database.

Result Set

QuOnto has a rewriting algorithm, so the result set is a set of tuples that satisfy the query. The result of the D2RQ mapping is a mapping file that can be mapped to a RDF vocabulary or an OWL model, and queried by SPARQL. The result of the mapping with RDBToOnto is a populated ontology in RDF/OWL where some tables in the database are merged into new structures, new categories by following patterns. The result of a mapping with RDB2ONTO is a RDF/OWL ontology model. A mapping with D2OMapper results in a XML document. VisAVis has a mapping of a kind. The tool creates an ontology and fills the property of the classes with a String that is a SQL query. The the query is executed against the database it returns the dataset for the class.

Tools	Requirements		
	Available	**Free of Charge**	**Documented**
QuOnto	For testing	YES	Some
D2RQ	YES	YES	YES
RDBToOnto	YES	YES	YES
RDB2ONTO	YES	YES	YES
D2OMapper	NO	Unknown	NO
VisAVis	YES	YES	Some

Table 4.2: Overview of the tools accessibility and documentation.

Tools	Solution
	Result-set
QuOnto	Set of tuples that satisfy the query
D2RQ	Mapping-file, that is possible to wrap into an ontology
RDBToOnto	A populated ontology in RDF/OWL
RDB2ONTO	RDF/OWL ontology model
D2OMapper	XML document of the mapping
VisAVis	Ontology where the class property is a SQL query that returns the dataset

Table 4.3: Overview of the tools result-set

Chapter 5

The Mapping Process

This chapter will start by looking at the different mapping rules for mapping from ORM to OWL, to provide the basic understanding of the process. It will be followed by a section that shows alternative ways of expressing certain structures in OWL. Last is a section that describes the finishing touch of a mapping process.

5.1 The Mapping Rules

Some preliminaries have been made regarding the ORM model used to demonstrate the mapping rules. All concepts are referential, i.e. they either have a perfect bridge or an external uniqueness that can be used to identify and represent its instances. In addition all non-perfect bridges have been rewritten into perfect bridges.

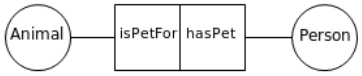
The mapping from ORM to OWL should produce only a T-box in OWL and no A-box (i.e. individuals). It is important to limit the OWL model to only this level because this is the level that expresses the kind of general statements that are comparable to the statements in an ORM model. When the A-box is included in an OWL model it is no longer a general model, but a model that specifies single cases. OWL has many constraints that only work on individuals.

The different structures in ORM and OWL become more apparent during a mapping. In ORM the "first class citizens", fact types and bridges, are the basic building blocks, while in OWL the basic building blocks are classes and properties. Since the mapping goes from ORM to OWL, we will start with explaining how the basic building blocks in ORM are mapped to OWL, and continue with the more specific mapping rules. In some of the mapping rules the OWL code takes up so much space that it has been divided into two columns. In these cases the left column describe the object properties and the right column describe the classes.

5.1.1 Mapping a Binary Fact Type from ORM to OWL

A binary fact type in ORM consists of two concepts that are connected with a binary relationship. Each of the two concepts is mapped to a class in OWL and each of the two roles in the binary relationship is mapped into an object property (Hodrob & Jarrar, 2010; Wagih,

ElZanfaly, & Kouta, 2011). The two object properties must be made inverse of one another, and this is done with `owl:inverseOf`. Since properties are independent of classes in OWL the object properties need to be restricted so that they apply only for the two classes they are supposed to connect. This is done by restricting their domain with `owl:domain` and their range with `owl:range`. Figure 5.1 show an example of this mapping rule.

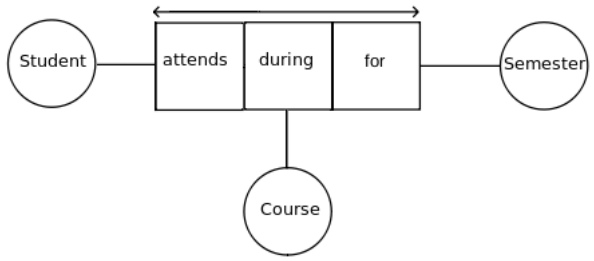



```
:Animal rdf:type owl:Class  .

:Person rdf:type owl:Class .

:isPetFor rdf:type owl:ObjectProperty ;
    owl:domain :Animal ;
    owl:range :Person ;
    owl:inverseOf :hasPet .

:hasPet rdf:type owl:ObjectProperty .
```

Figure 5.1: Mapping of a binary fact type from ORM to OWL.

5.1.2 Mapping N-ary Fact Types from ORM to OWL

ORM supports n-ary fact types, where the most commonly used ones are ternary and quaternary fact types. OWL, on the other hand, is only capable of creating an equal to binary fact types by using inverse properties. The statements constructed with n-ary fact types are possible to construct in OWL. An n-ary fact type in ORM can be rewritten to n binary fact types by adding a new concept which combines the n-ary relationship. This can be done in OWL by adding an extra class that combines the others. The top part of Figure 5.2 shows a ternary fact type in ORM. The concepts are mapped to classes and the roles are mapped to inverse object properties, just as it is done for a binary fact type. In order to combine the classes an extra class `CourseAttendance` is added. This class combines the ternary relationship between `Student`, `Course` and `Semester`. The classes are connected to `CourseAttendance` through object properties, and the combination of these object properties are unique for `CourseAttendance`. A key can be used to create this uniqueness. The bottom part of Figure 5.2 shows the OWL construction of the ternary fact type.



Object properties:

```
:studentAttendsCourse
rdf:type owl:ObjectProperty ;
owl:domain :Student ;
owl:range :CourseAttendance .

:courseAttendanceForStudent
rdf:type owl:ObjectProperty ;
owl:inverseOf
:studentAttendsCourse .

:courseAttendanceInSemester
rdf:type owl:ObjectProperty ;
owl:domain :CourseAttendance ;
owl:range :Semester .

:semesterForCourseAttendance
rdf:type owl:ObjectProperty ;
owl:inverseOf
:courseAttendanceInSemester .

:courseAttendanceForCourse
rdf:type owl:ObjectProperty ;
owl:domain :CourseAttendance ;
owl:range :Course .

:courseInCourseAttendance
rdf:type owl:ObjectProperty ;
owl:inverseOf :courseAttendanceForCourse .
```

Classes:

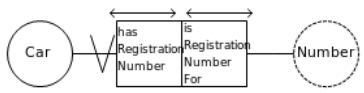
```
:Student rdf:type owl:Class .

:Course rdf:type owl:Class .

:Semester rdf:type owl:Class .

:CourseAttendance rdf:type owl:Class ;
owl:hasKey
(:semesterForCourseAttendance,
:courseInCourseAttendance,
:studentAttendsCourse ) .
```

Figure 5.2: Mapping of a ternary fact type from ORM to OWL.

5.1.3 Mapping Perfect Bridges from ORM to OWL

A perfect bridge is used to uniquely identify instances of a concept. A perfect bridge is when a concept and a value type are in a binary relationship with one another, and both of them can play their role only once. In addition the concept *must* play this role and is made unique by the value type. Figure 5.3 shows a perfect bridge where all instances of concept `Car` must play the role `hasRegistrationNumber` exactly once, and the instances of the value type `Number` have to be unique. A value type in ORM is mapped to a datatype property in OWL (Wagih et al., 2011). Since there are no inverse properties for datatype properties only one of the roles, `hasRegistrationNumber`, is represented as a datatype property. The lack of an inverse property makes it impossible to have a unique value for the property, so class `Car` can not have a sufficient property restriction with `hasRegistrationNumber`. The necessary uniqueness

is possible to express with a key (Wagih et al., 2011). In the figure class `Car` has the object property `hasRegistrationNumber` as a key.

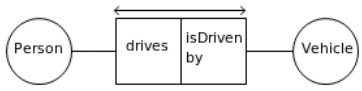



```
:hasRegistrationNumber
 rdf:type owl:DatatypeProperty ;
    rdfs:domain :Car .

:Car rdf:type owl:Class ;
   owl:hasKey (:hasRegistrationNumber) .
```

Figure 5.3: Mapping of a perfect bridge from ORM to OWL.

5.1.4 Internal Uniqueness that Span Both Roles in a Binary Relationship

An internal uniqueness that spans both roles in a binary relationship expresses that the combination of the instances that plays the roles can not be repeated. In a database this will be realized as a uniqueness constraint (key) over two columns, i.e. the pair of two values can not be repeated. In OWL the roles are mapped to inverse object properties which can have several different values, as long as all of them are individuals from a specified class. To restrict the values to be an individual from a specific class one can to use `rdfs:domain` and `rdfs:range`. Figure 5.4 shows that the object property `drives` has the class `Person` as domain and the class `Vehicle` as range, and that the object property `isDrivenBy` is made inverse of object property `drives`.




```
:driver rdf:type owl:ObjectProperty .;
    rdfs:domain :Person ;
    rdfs:range :Vehicle .

:isDrivenBy rdf:type owl:ObjectProperty
    owl:inverseOf :drives .
```

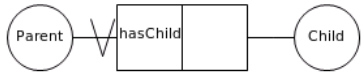
Figure 5.4: Mapping of an internal uniqueness that span both roles in a binary relationship from ORM to OWL.

Mandatory Role Constraint and Internal Uniqueness over a Single Role

Mandatory role constraint and internal uniqueness are common restrictions in ORM and are often combined. A mandatory role constraint expresses that an instance has to play the role, but it does not limit the number of times. An internal uniqueness constraint over a single role expresses that if an instance plays the role, it can not play it more than once.

5.1.5 Mandatory Role

Mandatory role constraint is mapped to the value constraint `owl:someValuesFrom` in OWL. Figure 5.5 shows an ORM structure where the concept `Parent` has a mandatory role constraint on its role `hasChild`. The constraint is mapped to a property restriction on the class `Parent` with the restriction `owl:someValuesFrom`. The constraint expresses that the object property `hasChild` must have at least one value that is an individual from class `Child`.

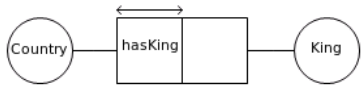



```
:Parent rdf:type owl:Class ;
   rdfs:subClassOf
[ rdf:type owl:Restriction ;
  owl:onProperty :hasChild ;
  owl:someValuesFrom :Child ] .

:hasChild rdf:type owl:ObjectProperty ;
 rdfs:domain :Parent ;
 rdfs:range :Child .
```

Figure 5.5: Mapping of a mandatory role constraint from ORM to OWL.

5.1.6 Internal Uniqueness over a Single Role

Internal uniqueness is mapped to the qualified cardinality constraint `owl:maxQualifiedCardinality` in OWL. Figure 5.6 shows an internal uniqueness that spans over the role `hasKing`. The cardinality constraint has the value `"1"^^xsd:nonNegativeInteger` which expresses that the object property `hasKing` may have at most 1 semantically distinct value. In addition the value has to be an individual from class `King`. The result is that the object property `hasKing` can have at most 1 value from class `King`.

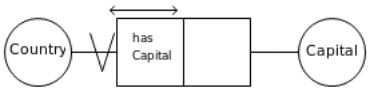



```
:Country rdf:type owl:Class ;
   rdfs:subClassOf
[ rdf:type owl:Restriction ;
  owl:onProperty :hasKing ;
  owl:onClass :King ;
  owl:maxQualifiedCardinality
  "1"^^xsd:nonNegativeInteger ] .

:hasKing rdf:type owl:ObjectProperty ;
 rdfs:domain :Country ;
 rdfs:range :King .
```

Figure 5.6: Mapping of a single internal uniqueness constraint from ORM to OWL.

5.1.7 Mandatory Role Constraint Combined with Single Role Internal Uniqueness Constraint

A mandatory role constraint combined with a single role internal uniqueness constraint on the same role in ORM expresses that an instance has to play the role exactly once. In OWL this can be constructed by using the qualified cardinality constraint `owl:qualifiedCardinality` with the value `"1"^^xsd:nonNegativeInteger`. Figure 5.7 shows a mandatory role and an internal uniqueness over the role `hasCapital`. The qualified cardinality constraint expresses that all individuals in class `Country` that have triples where the property `hasCapital` is used, must have exactly 1 semantically distinct value. In addition the value of the triple need to be an individual from class `Capital`.

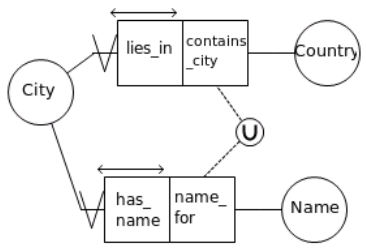



```
:Country rdf:type owl:Class ;
   rdfs:subClassOf
[ rdf:type owl:Restriction ;
  owl:onProperty :hasCapital ;
  owl:onClass :Capital ;
  owl:qualifiedCardinality
  "1"^^xsd:nonNegativeInteger ] .

:hasCapital rdf:type owl:ObjectProperty ;
 rdfs:domain :Country ;
 rdfs:range :Capital .
```

Figure 5.7: Mapping of a mandatory role with internal uniqueness constraint from ORM to OWL.

5.1.8 External Uniqueness

External uniqueness constraint between roles is, in addition to perfect bridges, used to uniquely identify instances of concepts. The combination of the roles touched by this constraint should be unique and not repeated for all the instances of the common concept. The ORM part of Figure 5.8 shows a concept `City` in relationships with the concepts `Country`, and `Name`. For all instances of the concept `City` the combination of the `Country` it lies in, and the `Name` it has should be unique. In OWL this uniqueness can be constructed with a key. The key is made up by the object properties `lies_in` and `has_name`. When one uses a key it is not necessary to additionally map the mandatory role or the internal uniqueness.




```
:City rdf:type owl:Class ;
   owl:hasKey (:lies_in, :has_name) .
```

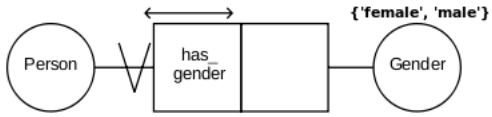
Figure 5.8: Mapping of an external uniqueness from ORM to OWL.

5.1.9 Value Constraint

A value constraint is a restriction on the population of a concept or a value type so that the value can only be one of a specific set of values, a value within a specific range, or both (Halpin et al., 2008). A value constraint in ORM is mapped to a class with a property restriction in OWL. The list of values should contain static values that are not likely to change in order to avoid changing the constraint (Hodrob & Jarrar, 2010; Wagih et al., 2011).

Value Constraint on a Concept

Figure 5.9 shows an ORM model with the concepts `Person` and `Gender` where the value of the gender can only be `male` or `female`. The role `hasGender` is mapped to an object property since the constraint is on a concept. In order to restrict the value of `hasGender` to be one of male or female, the property `owl:oneOf` is used as a restriction on class `Person`. The list of values that follows `owl:oneOf` is (female, male) which are the valid values for `hasGender`.

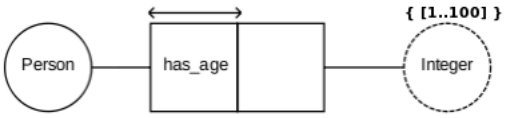



```
:hasGender rdf:type owl:ObjectProperty ;
 rdfs:range :Gender .

:Person rdf:type owl:Class ;
 rdfs:subClassOf
[ rdf:type owl:Restriction ;
  owl:onProperty :hasGender ;
  owl:onClass [ rdf:type owl:Class ;
  owl:oneOf ( :female :male ) ] ;
  owl:qualifiedCardinality
  "1"^^xsd:nonNegativeInteger ] .
```

Figure 5.9: Mapping of a value constraint from ORM to OWL.

Value Constraint on a Value Type

Figure 5.10 shows an ORM model with a concept `Person` in a relationship with a value type by the role `has_age`. The value restriction on the value type expresses that a person is only allowed to have an age between 1 and 100. It is mapped to a datatype property in OWL, since the constraint is on a value type (Wagih et al., 2011). The datatype property gets an integer as range since the valid values are numbers. The class `Person` has a property restriction on the datatype property `has_age` which is additionally restricted to have a data range between `xsd:minInclusive 1 and xsd:maxInclusive 100`.




```
:has_age rdf:type owl:DatatypeProperty ;
 rdfs:domain :Person ;
 rdfs:range xsd:integer .

:Person rdf:type owl:Class ;
        rdfs:subClassOf
[ rdf:type owl:Restriction ;
 owl:onProperty :has_age ;
owl:maxQualifiedCardinality
"1"^^xsd:nonNegativeInteger ;
 owl:onDataRange
[ rdf:type rdfs:Datatype ;
  owl:onDatatype xsd:integer ;
  owl:withRestrictions
( [ xsd:minInclusive 1 ]
  [ xsd:maxInclusive 100 ] ) ] ] .
```

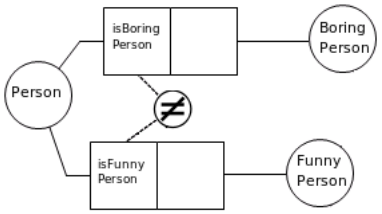
Figure 5.10: Mapping of a value constraint from ORM to OWL.

Mapping Set-Comparison Constraints from ORM to OWL

Set-comparison constraints restrict the way the population of one role, or several roles, is related to the population of another (Halpin et al., 2008). These constraints can limit the number of roles a concept can play and limit the values in concepts.

5.1.10 Exclusion Constraint

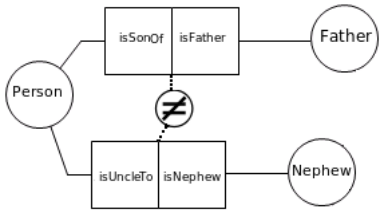
An exclusion constraint between single roles expresses that an instance can play *at most* one of the roles (Halpin et al., 2008). Figure 5.11 shows an exclusive constraint between single roles that is mapped to a property restriction on the class `Person`. It is necessary to get the complement of the intersection of the object properties `isBoringPerson` and `isFunnyPerson`. This can be done by using `owl:complementOf` and `owl:intersectionOf`.




```
:Person rdf:type owl:Class ;
 rdfs:subClassOf
[ rdf:type owl:Class ;
  owl:complementOf
[ rdf:type owl:Class ;
owl:intersectionOf
( [ rdf:type owl:Restriction ;
    owl:onProperty :isBoringPerson ;
    owl:allValuesFrom :Person ],
  [ rdf:type owl:Restriction ;
    owl:onProperty :isFunnyPerson ;
    owl:allValuesFrom :Person ] ) ] ] .
```

Figure 5.11: Mapping of an exclusion constraint on single roles from ORM to OWL.

An exclusion constraint between a pair of roles means that both instances in the pair can play at most one of the relationships. Figure 5.12 shows an ORM model with an exclusion constraint between the role pairs `isSon - isFather` and `isUncle - isNephew`. This constraint expresses that the two people that are in the son - father relationship can not at the same time be in the uncle - nephew relationship. In OWL this can be constructed by making the object property `isSonOf` disjoint from `isUncle` and the object property `isFather` disjoint from `isNephew`. The property `owl:propertyDisjointWith` (Wagih et al., 2011) is used to state this. When two properties are disjoint from each other it means that there can not exist two statements where the subject and object of each statement are the same for the object properties (Hebeler et al., 2009).



Object properties:

```
:isSonOf rdf:type owl:ObjectProperty ;
rdfs:domain :Person ;
rdfs:range :Father .

:isFather rdf:type owl:ObjectProperty ;
owl:inverseOf :isSonOf ;
owl:propertyDisjointWith :isNephew .

:isUncleTo rdf:type owl:ObjectProperty ;
rdfs:domain :Person ;
rdfs:range :Nephew ;
owl:propertyDisjointWith :isSonOf .

:isNephew rdf:type owl:ObjectProperty ;
owl:inverseOf :isUncleTo .
```

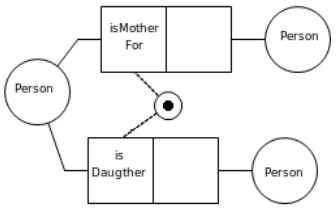
Classes:

```
:Person rdf:type owl:Class .

:Father rdf:type owl:Class .

:Nephew rdf:type owl:Class .
```

Figure 5.12: Mapping of an exclusion constraint on role pairs from ORM to OWL.

5.1.11 Mandatory Constraint

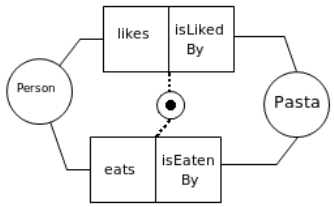
A mandatory constraint (i.e. inclusive-or constraint) between single roles expresses that an instance of a concept must play *at least* one of the associated roles. In OWL this constraint is constructed by using `owl:unionOf` (Hodrob & Jarrar, 2010; Wagih et al., 2011) to make a union of the individuals playing the two roles. In Figure 5.13 this property expresses that the set of all people are the union of the people who are mothers or daughters or both. The class `Person` is a subclass of a property restriction with a union of the object properties `isMotherFor` and `isDaugther`.




```
:Person rdf:type owl:Class ;
rdfs:subClassOf
[ rdf:type owl:Class ;
 owl:unionOf (
[ rdf:type owl:Restriction ;
  owl:onProperty :isMotherFor ;
  owl:allValuesFrom :Person ]
[ rdf:type owl:Restriction ;
  owl:onProperty :isDaugther ;
  owl:allValuesFrom :Person ] ) ] ;
```

Figure 5.13: Mapping of an mandatory constraint on single roles from ORM to OWL.

A mandatory constraint between a pair of roles means both instances in the pair must play *at least* one of the roles. This constraint is also mapped to OWL with `owl:unionOf`, the only different is that it needs to be stated for both classes in the relationship. Figure 5.14 shows the classes `Person` and `Pasta` with the union of the object properties as a property restriction.



Object properties:

```
:Person rdf:type owl:Class ;
rdfs:subClassOf
[ rdf:type owl:Class ;
 owl:unionOf (
[ rdf:type owl:Restriction ;
  owl:onProperty :likes ;
  owl:allValuesFrom :Pasta ]
[ rdf:type owl:Restriction ;
  owl:onProperty :eats ;
  owl:allValuesFrom :Pasta ] ) ] ;
```

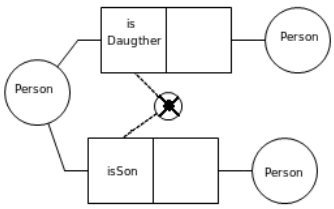
Classes:

```
:Pasta rdf:type owl:Class ;
rdfs:subClassOf
[ rdf:type owl:Class ;
 owl:unionOf (
[ rdf:type owl:Restriction ;
  owl:onProperty :isLikedBy ;
  owl:allValuesFrom :Person ]
[ rdf:type owl:Restriction ;
  owl:onProperty :isEatenBy ;
  owl:allValuesFrom :Person ] ) ] ;
```

Figure 5.14: Mapping of a mandatory constraint on role pairs from ORM to OWL.

5.1.12 Exclusive-Or Constraint

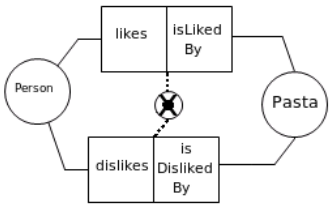
An exclusive-or constraint between single roles expresses that an instance of a concept must play *exactly* one of the roles. This constraint also requires getting the complement of the intersection between the roles (Wagih et al., 2011). The difference from an exclusion constraint is that the combination of the instances in the roles constitutes the entire concept. Figure 5.15 shows this constraint mapped to a disjoint union of the property restrictions on the object properties `isDaughter` and `isSon` by using `owl:disjointUnionOf`. In addition the object properties must be made disjoint of one another, so an individual can only have one of the object properties.




```
:isDaughter rdf:type owl:ObjectProperty ;
 rdfs:domain :Person ;
 rdfs:range :Person .

:isSon rdf:type owl:ObjectProperty ;
 rdfs:domain :Person ;
 rdfs:range :Person ;
 owl:propertyDisjointWith :isDaughter .

:Person rdf:type owl:Class ;
 rdfs:subClassOf
 [ rdf:type owl:Class ;
   owl:disjointUnionOf (
   [ rdf:type owl:Restriction ;
     owl:onProperty :isDaughter ;
     owl:allValuesFrom :Person ],
   [ rdf:type owl:Restriction ;
     owl:onProperty :isSon ;
     owl:allValuesFrom :Person ] ) ] ;
```

Figure 5.15: Mapping of an exclusive-or constraint on single roles from ORM to OWL.

An exclusive-or constraint between a pair of roles expresses that both instances in the pair must play *exactly* one of the roles. The combination of the instances in the pair can only be in one of the relationships. The constraint on a role pair is mapped the same way as the constraint on single roles only that the restrictions need to be on both classes in the relationship. Figure 5.12 shows an exclusive-or constraint on a pair of roles and how it is mapped to OWL.



Object properties:

```
:likes rdf:type owl:ObjectProperty ;
 rdfs:domain :Person ;
 rdfs:range :Pasta .

:isLikedBy rdf:type owl:ObjectProperty ;
owl:inverseOf :likes .

:dislikes rdf:type owl:ObjectProperty ;
 rdfs:domain :Person ;
 rdfs:range :Pasta ;
 owl:propertyDisjointWith :likes .

:isDislikedBy rdf:type owl:ObjectProperty ;
owl:inverseOf :dislikes ;
owl:propertyDisjointWith :isLikedBy .
```

Classes:

```
:Person rdf:type owl:Class ;
 rdfs:subClassOf
 [ rdf:type owl:Class ;
   owl:disjointUnionOf (
   [ rdf:type owl:Restriction ;
     owl:onProperty :likes ;
     owl:allValuesFrom :Pasta ]
   [ rdf:type owl:Restriction ;
     owl:onProperty :dislikes ;
     owl:allValuesFrom :Pasta ] ) ] ;

:Pasta rdf:type owl:Class ;
 rdfs:subClassOf
 [ rdf:type owl:Class ;
   owl:disjointUnionOf (
   [ rdf:type owl:Restriction ;
     owl:onProperty :isLikedBy ;
     owl:allValuesFrom :Person ]
   [ rdf:type owl:Restriction ;
     owl:onProperty :isDislikedBy ;
     owl:allValuesFrom :Person ] ) ] ;
```

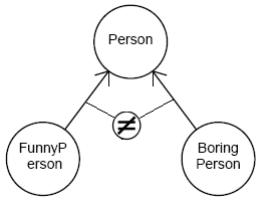
Figure 5.16: Mapping of an exclusive-or constraint on a pair of roles from ORM to OWL.

Subtype Constraints

A concept is a subtype when it is a specialization of another concept. There are three different types of subtype constraints: exclusive subtypes, exhaustive subtypes and partition constraint.

5.1.13 Exclusive Subtypes

Exclusive subtypes, also known as disjoint subtypes, expresses that the instances of the supertype is a member of *at most* one of the subtypes. In OWL it is constructed with `rdfs:subClassOf` and `owl:disjointClasses` (Hodrob & Jarrar, 2010; Wagih et al., 2011). When two classes are disjoint no individual can be a member of both classes. Figure 5.17 shows that the exclusive subtypes `FunnyPerson` and `BoringPerson` is mapped to subclasses of `Person`, and made disjoint from one another.

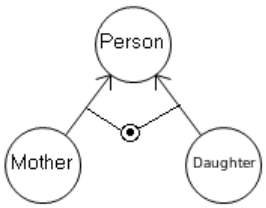



```
:FunnyPerson
 rdf:type owl:Class ;
 rdfs:subClassOf :Person .

:BoringPerson
 rdf:type owl:Class ;
 rdfs:subClassOf :Person ;
 owl:disjointWith :FunnyPerson .
```

Figure 5.17: Mapping of exclusive subtypes from ORM to OWL.

5.1.14 Exhaustive Subtypes

Exhaustive subtypes, also known as total constraint or inclusive-or constraint, expresses that the instances of the supertype *must* belong to *at least* one, but optionally several of the subtypes. In OWL this is constructed with `rdfs:subClassOf` and `owl:unionOf` (Hodrob & Jarrar, 2010; Wagih et al., 2011). Figure 5.18 shows that the exhaustive subtypes `Mother` and `Daughter` are mapped to subclasses of class `Person` and made into a union that is an equivalent class to `Person`.

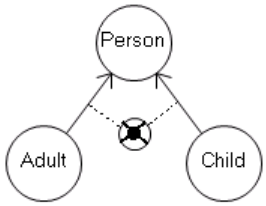



```
:Daughter rdf:type owl:Class ;
        rdfs:subClassOf :Person .

:Mother rdf:type owl:Class ;
        rdfs:subClassOf :Person .

:Person rdf:type owl:Class ;
owl:equivalentClass [ rdf:type owl:Class ;
owl:unionOf ( :Mother :Daughter ) ] ;
```

Figure 5.18: Mapping of an exhaustive subtypes from ORM to OWL.

5.1.15 Partition Constraint

Partition constraint, also known as exclusive-or, expresses that each instance of the supertype belongs to *exactly* one of the subtypes. In OWL this is mapped with `rdfs:subClassOf` and `owl:disjointUnionOf` (Wagih et al., 2011). Figure 5.19 shows that the partition constraint on the subtypes `Adult` and `Child` is mapped to a disjoint union with the property `owl:disjointUnionOf` and made an equivalent class of class `Person`.




```
:Adult rdf:type owl:Class ;
       rdfs:subClassOf :Person .

:Child rdf:type owl:Class ;
       rdfs:subClassOf :Person .

:Person rdf:type owl:Class ;
owl:equivalentClass
[ rdf:type owl:Class ;
  owl:disjointUnionOf
( :Adult :Child ) ] ;
```

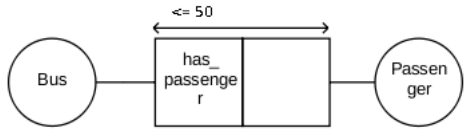
Figure 5.19: Mapping of a partition constraint from ORM to OWL.

5.1.16 Equivalence-Of-Path

Equivalence-Of-Path in ORM is used when there are two paths to one concept, and both paths leads to the same instance (Skagestein & Normann, 2008). This constraint is not possible to construct in OWL, and the consequences will be discussed in detail in Chapter 8.

5.1.17 Cardinality Constraint

Cardinality constraint, or frequency constraint, on a single role expresses how many times an instance of a concept can play the role. In OWL this constraint is constructed with a choice of `owl:minQualifiedCardinality`, `owl:qualifiedCardinalty` and `owl:maxQualifiedCardinality` (Hodrob & Jarrar, 2010; Wagih et al., 2011). Figure 5.20 shows the concept `Bus` that is restricted to have no more than 50 passengers. In OWL this is constructed with `owl:maxQualifiedCardinality` which expresses that all individuals of class `Bus` can have no more than `50` semantically distinct values in the object of triples where the property `hasPassenger` is used.

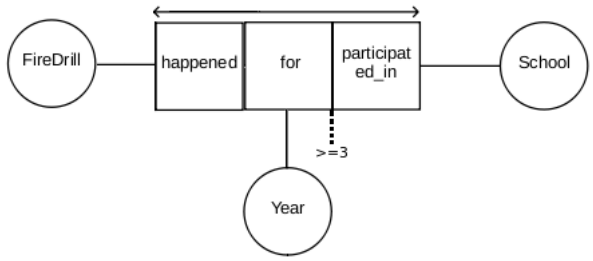



```
:Bus rdf:type owl:Class ;
  rdfs:subClassOf
[ rdf:type owl:Restriction ;
  owl:onProperty :hasPassenger ;
  owl:onClass :Passenger ;
  owl:maxQualifiedCardinality
  "50"^^xsd:nonNegativeInteger ] .
```

Figure 5.20: Mapping of a cardinality constraint from ORM to OWL.

5.1.18 Internal Multi-Role Cardinality Constraint

An internal multi-role cardinality constraint expresses that the combination of the instances playing the roles touched by this constraint must play the roles at least, at most or exactly n times. There exists no equal constraint to this in OWL, but the statements in a construction like this are still possible to express. Figure 5.21 shows a concept `FireDrill` that must happen at least 3 times per `Year` for each `School`. One way to express this statement in OWL is to add an extra class, `SchoolYear`, that combines the classes `School` and `Year` that are touched by the constraint. `SchoolYear` can get a single cardinality constraint `owl:minQualifiedCardinality` with the value 3 on the object property `SYforFD` (`SchoolYear_for_FireDrill`). This will ensure that there are at least three different values from class `FireDrill`. In addition the combination of `School` and `Year` should be made unique for `SchoolYear` by a key. Figure 5.21 shows how this construction looks like in OWL.



Object properties:

```
:happenedAtSY rdf:type
 owl:ObjectProperty ;
 rdfs:domain :Firedrill ;
 rdfs:range :SchoolYear ;
 owl:inverseOf SYforFD .
:SYforFD rdf:type
 owl:ObjectProperty .

:SYhasSchool rdf:type
 owl:ObjectProperty ;
 rdfs:domain :SchoolYear ;
 rdfs:range :School ;
 owl:inverseOf isSchoolForSY .
:isSchoolForSY rdf:type
 owl:ObjectProperty .

:SYhasYear rdf:type
 owl:ObjectProperty ;
 rdfs:domain :SchoolYear ;
 rdfs:range :Year ;
 owl:inverseOf yearForSY .
:yearForSY rdf:type
 owl:ObjectProperty .
```

Classes:

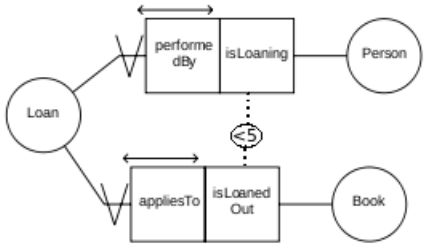
```
:Firedrill rdf:type owl:Class ;
 rdfs:subClassOf
[ rdf:type owl:Restriction ;
  owl:onProperty :happenedAt ;
  owl:someValuesFrom :SchoolYear ] .

:SchoolYear rdf:type owl:Class .
 rdfs:subClassOf
[ rdf:type owl:Restriction ;
  owl:onProperty :SYforFD ;
  owl:onClass :FireDrill ;
  owl:minQualifiedCardinality
  "3"^^xsd:nonNegativeInteger ] .
owl:hasKey (:SYhasSchool, :SYhasYear) .

:School rdf:type owl:Class .

:Year rdf:type owl:Class .
```

Figure 5.21: Mapping of multi-role cardinality constraint from ORM to OWL.

5.1.19 External Cardinality Constraint

An external cardinality constraint, or external frequency constraint, between roles expresses that the combination of the instances that plays the roles need to play them at least, at most or exactly *n* times for the common concept they are in a relationship with (Halpin et al., 2008). OWL has no constraint that can express this, but the statements in a construction like this are still possible to express. Figure 5.22 shows a structure in ORM with an external cardinality constraint between the concept `Person` and `Book`. The constraint expresses that a certain person can loan a certain book no more than 4 times. This statement can be constructed in the same way as a multi-role cardinality constraint, by adding an extra class that combines the two classes touched by the constraint. For the example in Figure 5.22 a class `LoanCase` can be added that combines `Person` and `Book`. The classes `Person` and `Book` must be made unique for the `LoanCase` so one loan case consists of one person and one book. This uniqueness can be expressed by a key. In addition class `LoanCase` gets a single cardinality constraint on its relationship with class `Loan` by `owl:maxQualifiedCardinality` with the value 4. This constraint will ensure that a loan case can have relationships with 4 different loans, meaning

the same loan case can be lent 4 times.



Object properties:

```
:loanAppliesTo rdf:type
owl:ObjectProperty ;
rdfs:domain :Loan ;
rdfs:range :LoanCase
owl:inverseOf :loanCaseConstitueLoan .

:loanCaseConstitueLoan rdf:type
owl:ObjectProperty .

:loanCasePerformedByPerson rdf:type
owl:ObjectProperty ;
rdfs:domain :LoanCase ;
rdfs:range :Person ;
owl:inverseOf :personIsLoaning .

:personIsLoaning rdf:type
owl:ObjectProperty .

:loanCaseAppliesToBook rdf:type
owl:ObjectProperty ;
rdfs:domain :LoanCase ;
rdfs:range :Book ;
owl:inverseOf :bookIsLoanedOut .

:bookIsLoanedOut rdf:type
owl:ObjectProperty .
```

Classes:

```
:Loan rdf:type owl:Class ;
[ rdf:type owl:Restriction ;
  owl:onProperty :loanAppliesTo ;
  owl:onClass :LoanCase ;
  owl:qualifiedCardinality
  "1"^^xsd:nonNegativeInteger ] .

:LoanCase rdf:type owl:Class ;
[ rdf:type owl:Restriction ;
  owl:onProperty :loanCaseConstituteLoan ;
  owl:onClass :Loan ;
  owl:maxQualifiedCardinality
  "4"^^xsd:nonNegativeInteger ] ;
owl:hasKey (:loanCasePerformedByPerson,
            :loanCaseAppliesToBook) .

:Person rdf:type owl:Class .

:Book rdf:type owl:Class .
```

Figure 5.22: Mapping of external cardinality constraint from ORM to OWL.

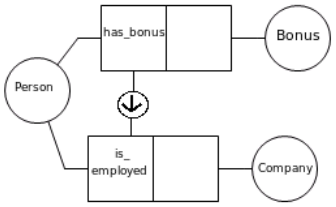
Set-Comparison Constraints

Set-comparison constraints define how the instances of concepts that play one role are related to the instances that play another role. There are two types of constraints on sets, namely subset and equality.

5.1.20 Subset Constraint

Subset constraint between roles expresses that an instance can not play the role at the tail of the arrow unless it also plays the role at the head of the arrow. This structure state that there are two sets; the set with instances that play the role at the tail of the arrow, which is a subset of the set of instances that play the role at the head of the arrow.

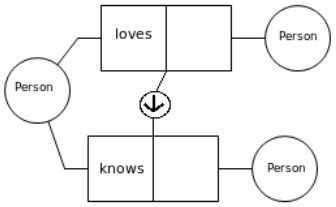
The constraint between single roles can be mapped to subclasses in OWL by using `owl:equivalentClass` and `owl:subClassOf`. The sets will be captured in two classes, which are subclasses of the original class. One class has the object property that is equal to the role at the tail of the arrow as an equivalent class, and the other has the object property equal to the role at the head of the arrow as an equivalent class. The class with the role from the tail of the arrow becomes a subclass of the class with the role from the head of the arrow. Figure 5.23 shows an example of a subset constraint between the single roles `has_bonus` and `is_employed`. In OWL this is mapped to the classes `PersonWithBonus` with the object property `hasBonus` which is a subclass of `PersonIsEmployed` with the object property `isEmployed` (Hodrob & Jarrar, 2010).




```
:PersonIsEmployed rdf:type owl:Class ;
owl:equivalentClass
[ rdf:type owl:Restriction ;
  owl:onProperty :is_employed ;
  owl:allValuesFrom :Company ] ;
rdfs:subClassOf :Person .

:PersonWithBonus rdf:type owl:Class ;
owl:equivalentClass
[ rdf:type owl:Restriction ;
  owl:onProperty :has_bonus ;
  owl:allValuesFrom :Bonus ] ;
rdfs:subClassOf :PersonIsEmployed .
```

Figure 5.23: Mapping of sub-set constraint between single roles from ORM to OWL.

The subset constraint between a pair of roles can be mapped to OWL by making the object property at the tail of the arrow a subproperty of the object property at the head of the arrow with `owl:subPropertyOf` (Hodrob & Jarrar, 2010; Wagih et al., 2011). Figure 5.24 shows an example of a subset constraint between two role pairs that is mapped with `owl:subPropertyOf`.




```
:knows rdf:type owl:ObjectProperty ;
rdfs:domain :Person;
rdfs:range :Person .

:loves rdf:type owl:ObjectProperty ;
owl:subPropertyOf :knows .
```

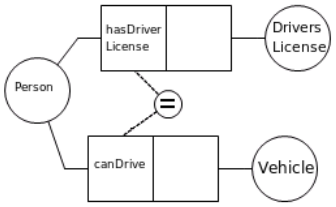
Figure 5.24: Mapping of sub-set constraint between role pairs from ORM to OWL.

5.1.21 Equality Constraint

Equality constraint expresses that two sets are equal to each other, meaning all instances that play one role must also play the other, and vice versa.

The equality constraint on single roles expresses that the set of instances that plays the roles touched by the constraint must be equal. In the same way as with subsets this constraint need to be mapped into subclasses in OWL in order to express the same. Figure 5.25 shows the constraint between the roles `hasDriversLicense` and `canDrive` which expresses that every person that has a drivers license can drive a vehicle, and vise versa. In OWL this is constructed by creating the class `PersonHasDriversLicense` with the object property `hasDriversLicense` as

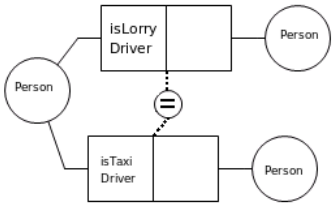
an equivalent class, and the class `PersonIsAllowedToDrive` with the object property `canDrive` as an equivalent class. In addition the classes need to be subclasses of `Person` and equivalent with each other.




```
:PersonHasDriversLicense
rdf:type owl:Class ;
owl:equivalentClass
[ rdf:type owl:Restriction ;
  owl:onProperty :hasDriversLicense ;
  owl:allValuesFrom :DriversLicense ] ,
:PersonCanDrive ;
rdfs:subClassOf :Person .

:PersonIsAllowedToDrive
rdf:type owl:Class ;
owl:equivalentClass
[ rdf:type owl:Restriction ;
  owl:onProperty :canDrive ;
  owl:allValuesFrom :Vehicle ] ,
:PersonHasDriversLicense ;
rdfs:subClassOf :Person .

:Person rdf:type owl:Class .
```

Figure 5.25: Mapping of equality constraint on single roles from ORM to OWL.

The equality constraint between a pair of roles in ORM can be mapped to OWL by making one object property equivalent to the other with `owl:equivalentObjectProperty` (Hodrob & Jarrar, 2010; Wagih et al., 2011). Figure 5.26 show the equality constraint mapped to two object properties `isTaxiDriver` and `isLorryDriver` which are equivalent to each other so that every `Person` that is a lorry driver also is a taxi driver, and vice versa.



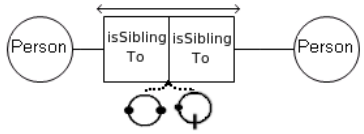

```
:isLorryDriver rdf:type
owl:ObjectPropoerty ;
rdfs:domain :Person .

:isTaxiDriver rdf:type
owl:ObjectPropoerty ;
rdfs:domain :Person ;
owl:equivalentProperty :isLorryDriver .
```

Figure 5.26: Mapping of equality constraint on role pairs from ORM to OWL.

5.1.22 Ring Constraints

A ring constraint is a constraint that can be put on role pairs where both roles are played by the same concept (Halpin et al., 2008). The ring constraints are symmetry, asymmetry, reflexivity, irreflexivity and transitivity.

- A relation of type R is **symmetric** iff for all x, y xRy → yRx (Halpin et al., 2008).
- A relation of type R is **asymmetric** iff for all x,y xRy → ¬yRx (Halpin et al., 2008).




```
:isSiblingTo rdf:type
 owl:ObjectProperty ,
 owl:IrreflexiveProperty ,
 owl:SymmetricProperty ;
rdfs:domain :Person ;
rdfs:range :Person .
```

Figure 5.27: Mapping of a ring constraint from ORM to OWL

```
:Parent rdf:type owl:Class ;
   rdfs:subClassOf
[ rdf:type owl:Restriction ;
  owl:onProperty :hasChild ;
  owl:minCardinality
"1"^^xsd:nonNegativeInteger ] .
```

```
:Parent rdf:type owl:Class ;
   rdfs:subClassOf
[ rdf:type owl:Restriction ;
  owl:onProperty :hasChild ;
  owl:onClass :Child ;
  owl:minQualifiedCardinality
"1"^^xsd:nonNegativeInteger ] .
```

Figure 5.28: Alternative mapping of a mandatory role from ORM to OWL

- A relation of type R is **reflexive** over its population iff for each possible population of R, xRx for each x in pop(r1) ∪ pop(r2) (Halpin et al., 2008).
- A relation R is **irreflexive** iff for all x ¬xRx (Halpin et al., 2008).
- A relation of type R is **transitive** iff for all x, y, z xRy and yRz → xRz (Halpin et al., 2008).

In OWL ring constraints are specific property classes that are subclasses of `owl:ObjectProperty` (Hebeler et al., 2009). The names of the property classes are `owl:SymmetricProperty`, `owl:AsymmetricProperty`, `ReflexiveProperty`, `owl:IrreflexiveProperty` and `owl:TransitiveProperty`. Figure 5.27 shows the ring constraints irreflexive and symmetric on the role `isSiblingTo` that connects two people. In OWL this is mapped with `owl:IrreflexiveProperty` and `owl:SymmetricProperty` so that a person can not be a sibling to oneself and the one you are a sibling to is also your sibling.

5.2 Alternative Constructions

There are in general several ways of expressing the same logical model in OWL. This section will give some examples of different ways of constructing certain constraints than the ones presented in the previous section.

5.2.1 Mandatory Role

A mandatory role constraint can, in addition to `owl:someValuesFrom`, also be constructed by using `owl:minCardinality` with the value `"1"^^xsd:nonNegativeInteger` which expresses that there has to be at least 1 semantically distinct value for the property. The second way of constructing this constraint is by using `owl:minQualifiedCardniality` with the value `"1"^^xsd:nonNegativeInteger`. This constraint limits not only the values of the property to be minimum 1 distinct value, but it also requires that the value is from the specified class or data range. Figure 5.28 shows the alternative ways of mapping a mandatory role.

```
:hasKing rdf:type owl:FunctionalProperty ,
             owl:ObjectProperty .
```

Figure 5.29: Alternative mapping of a single internal uniqueness from ORM to OWL

```
:Person rdf:type owl:Class ;
   rdfs:subClassOf
 [ rdf:type owl:Restriction ;
   owl:onProperty :drives ;
   owl:allValuesFrom :Vehicle ] .

:Vehicle rdf:type owl:Class ;
   rdfs:subClassOf
  [ rdf:type owl:Restriction ;
   owl:onProperty :isDrivenBy ;
   owl:allValuesFrom :Person ] .
```

Figure 5.30: Alternative mapping of an internal uniqueness that spans both roles from ORM to OWL

5.2.2 Internal Uniqueness

The internal uniqueness can, in addition to `owl:maxQualifiedCardinality`, also be constructed with `owl:FunctionalObjectProperty` or with `owl:maxCardinality` with restriction 1 integer (Hodrob & Jarrar, 2010; Wagih et al., 2011). Figure 5.29 shows how a functional property is created.

5.2.3 Internal Uniqueness that Span Both Roles in a Binary Relationship

The internal uniqueness that spans both roles in a binary relationship can be restricted to the class as well as the object property. It can be restricted as a property restriction on the class with the property `owl:allValuesFrom` as seen in Figure 5.30 with class `Person` and `Vehicle`. This property restriction will not prevent other classes from using the property `drives` or `isDrivenBy`, but it will restrict the properties to only to have values from class `Vehicle` and `Person` for individuals in class `Vehicle` and `Person`. Figure 5.30 shows the alternative way of mapping an internal uniqueness that spans both roles in a binary relationship.

5.2.4 Value Constraint on Object Property

In OWL restriction on the values of an object property can be constructed on the object property as well as on the class, by using `rdfs:range`. The range can have as value an anonymous class consisting of `owl:oneOf` with a list of values that are individuals. The list may be formed with or without ranges and only the elements in the list are valid values for the property. Figure 5.31 shows a value restriction on the object property `hasGender` that is restricted to be either `female` or `male`.

```
:hasGender rdf:type owl:ObjectProperty ;
rdfs:range [ rdf:type owl:Class ;
owl:oneOf ( :female :male) ] .
```

Figure 5.31: Alternative mapping of a value constraint on an object property from ORM to OWL

```
:has_age rdf:type owl:DatatypeProperty ;
 rdfs:range
[ rdf:type rdfs:Datatype ;
owl:onDatatype xsd:integer ;
owl:withRestrictions
([xsd:minInclusive 1]
[xsd:maxInclusive 100] ) ] .
```

Figure 5.32: Alternative mapping of a value constraint on a datatype property from ORM to OWL

5.2.5 Value Constraint on Datatype Property

In OWL restriction on the values of a datatype property can be constructed on the datatype property as well as on the class, with `rdfs:range`. The range can have as value a restricted anonymous class. This restricted class is of type `Datatype` and has additionally restrictions on a `xsd:integer` with `owl:withRestrictions`. The value of `owl:withRestrictions` is an integer between 1 and 100. Figure 5.32 shows a value restriction on the datatype property `has_age` that is restricted to be an integer between 1 and 100.

5.3 Finishing Touch

The Non Unique Name Assumption (NUNA) in OWL makes it necessary to make the classes in the OWL model disjoint. When modeling with NUNA names are not unique and there is no way of knowing that classes with different names are different from each other. In OWL a class describes a type of individuals that share some common characteristics, which allows individuals that are semantically the same to belong to different classes. In ORM all concepts contain different instances, with the exception of subtypes. In order for the OWL classes mapped from ORM to populate in the same manner as concepts in ORM it is necessary to make all the classes separate, with the exception of subclasses.

Chapter 6

The Norwegian National Register Case

This chapter will begin with a quick introduction to the Norwegian National Register, its work and function. Next is a section that explains the statements that form the foundation for the ORM model. It is followed by a description of how the ORM model over the Norwegian National Register was created. Last is a detailed description of the mapping from the ORM model to the OWL model based on the mapping rules presented in Chapter 5.

6.1 The Norwegian National Register

The Norwegian National Register (NOR: det sentrale folkeregisteret, dsf) is an official registry of all people who currently live in, or have been living in Norway (Folkeregisteret, 2011). It is one of the government institutions that participate in the Semicolon II project. The Norwegian National Register currently has all of its data stored in databases and desires an easier way to share it with the other institutions, than the current one.

The National Register stores information about, among others, the following (Folkeregisteret, 2011):

- Person's date of birth, identification number and names
- Parenthood and parental responsibility
- Relocation within Norway and abroad
- Marital statuses
- Deaths
- Citizenship
- Adoptions
- Residence and work permits

Some of the responsibilities of the Norwegian National Register are to check that marriages are valid, to check that name changes are valid, and assigning permanent and temporary identification numbers to people.

1. Every person must have at least one first name, may have several middle names and must have exactly one last name.
2. Every person must have a date of birth.
3. Every date of birth consists of a day within a month, a month and a year of birth.
4. Every person must have a unique identification, which is either a D-number or a birth number.
5. Every person must have a place of birth.
6. Every person must have a marital status.
7. Every person must have a gender.
8. Every person may have an address.
9. Every person may have a mother and a father, but can not be his/her own parent.
10. Every person that has a mother may have a comother, but she can not be the same person.
11. Every person may have one or several children.
12. Every marriage consists of two different people, and a person can only be married to one other person at the time. The one you are married to are also married to you.
13. Every relocation must be reported in a relocation notice, and applies to exactly one person and one point in time. For a given time and person there should be only one relocation.
14. Every relocation notice may involve relocation of people.
15. Every relocation notice must have a moving-away address, a moving-to address, a registration time and a time for the relocation.
16. Every person that is concerned by a relocation notice has the same relocation date.
17. Every address must lie in a country.
18. Every address must belong to an election district.
19. Every address may lie in a school district.
20. Every address may lie in a church parish.
21. Every address may lie in a part of town.

Figure 6.1: The statements that has been modeled.

6.2 The Statements

The ORM model is inspired by the Norwegian National Register. The aim is to present a model that is sufficiently complex to show the differences in expressions in the two modeling languages, ORM and OWL. We have included just enough information to demonstrate the similarities and differences between the languages, without having a model of such size that it overshadows the purpose of the modeling. It was not possible to get hold of the schemas for the database of the Norwegian National Register, so the ORM model is inspired by the Norwegian National Registers handbook, the different forms from their website (Skatteetaten, 2011), their rules and laws and their user manual.

The starting point has been to model the most basic information stored in the Norwegian National Register. Some parts of the model have been simplified in lack of concrete knowledge of the structure, or lack of insight into the prevailing rules and regulations. In addition most of the notification forms and the user manuals are very implementation specific. One concept that has been simplified is address, since it seems fairly complex. There are (at least) two kinds of addresses, location-addresses and street-addresses. Location addresses are used for farms and remote areas, while street-addresses are used for the rest. A location-address has a location name, a farm number, a subnumber and an optional attachment number just to mention some. A street-address has a street name, house number, house letter and a street code. Both kinds of addresses have a resident number. In order to avoid confusion and errors an address is simply represented as a textual description in the model.

The Norwegian National Register contains much more information about a person than what

is included in the model presented here. The main focus has been people and relationships between people, their address, names, gender and identification. The relationships described are those within a family, parenthood and marriage. The concept of sibling is not included explicitly. An included concept that may not be known to all is comother (NOR: medmor). It is used to describe a female that is the wife or cohabitant of a child's mother. A rule that will not be included is that every person *must* have a mother and a father. Because of the Closed World Assumption it is very impractical to have an absolute rule like this since one will be forced to add grandparents and great grandparents and so on ad infinitum. Some additional rules about a person have been modeled, such as marital status and place of birth. The rules of relocation notice have been included to demonstrate the possible complexity of the model.

The data in the model reflects the present time, so data concerning the past, e.g. previous marriages and deceased relatives, have been excluded. It will increase the complexity of the model without illustrating any more of the languages capabilities. Figure 6.1 shows the statements that is the basis for the modeling.

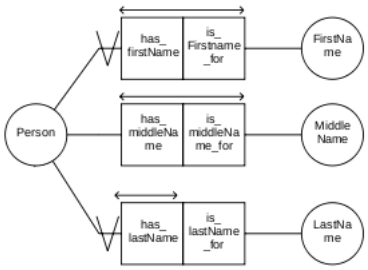
6.3 Creating the ORM model

The modeling was done by following the Conceptual Schema Design Procedure's seven step and Skagestein's rules of thumb (Halpin et al., 2008; Skagestein, 1996). This section will go through the statements in Figure 6.1 and explain what kind of structure they result in and why.

The tools chosen for modeling ORM is stORM. This tool is used in database courses at the University of Oslo, which makes it familiar from previous work and therefore easy to access and use. Familiarity and available experts on the tool has taken precedence over support of constructions and constraints. In addition it is free of charge. The tool chosen for modeling OWL is Protégé. It is also familiar from previous work and there are available experts on the use of it at the University of Oslo. It is free of charge and supports several different syntaxes for the output file. In addition the output file is not proprietary, easy to read and quick to get an overview over.

Statement 1 - Person with Name

Statement 1 in Figure 6.1 expresses that every person must have at least one first name, may have several middle names and must have exactly one last name. This statement results in four concepts: `Person`, `FirstName`, `MiddleName` and `LastName`. All the name concepts will be in a binary relationship to the concept `Person` since people are the ones having names and the different types of names are names for people. The restrictions on the number of names are constructed with mandatory role constraints and internal uniqueness constraints. The roles `has_firstName` and `has_lastName` have a mandatory role constraint since they are non optional. The relationship between `Person` and `FirstName` and `Person` and `MiddleName` has a long internal uniqueness indicating that the combination of the roles can not be repeated. That means that a person may have more than one first name and middle name, but not the same first or middle name twice. Figure 6.2 shows what this statement is modeled to in ORM.

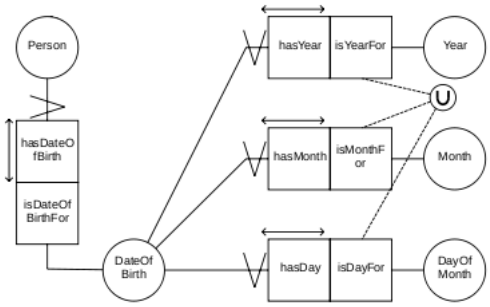


1. Every person must have at least one first name, may have several middle names and must have exactly one last name.

Figure 6.2: Statement 1 modeled in ORM.

Statement 2 and 3 - Person with Date of Birth

Statement 2 in Figure 6.1 expresses that every person must have a date of birth. Statement 3 in the same Figure states that a date of birth consists of a day within a month, a month and a year. It is common knowledge that a person only has one date of birth, but that several people may share the same date of birth. These statements results in four concepts: `Person`, `DateOfBirth`, `DayOfMonth`, `Month` and `Year`. The concept `Person` will be in a binary relationship with the concept `DateOfBirth`. `DateOfBirth` will be in binary relationships with the three remaining concepts, since a date of birth consists of a day, month and year. The combination of the roles played by `DayOfMonth`, `Month` and `Year` will be made unique for every `DateOfBirth` by an external uniqueness constraint. The roles from `DateOfBirth` to `Year`, `Month` and `DayOfMonth` are given a mandatory role constraint and a single internal uniqueness to ensure that a date of birth only plays these roles once. The role from `Person` to `DateOfBirth` is also given a mandatory role constraint and a single internal uniqueness to ensure that a person can not play the role more than once. Figure 6.3 shows what these statements are modeled to in ORM.



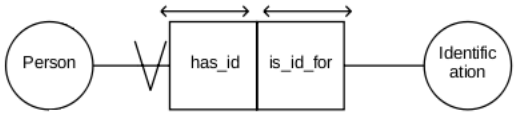
2. Every person must have a date of birth.
3. A date of birth consists of a day within a month, a month and a year.

Figure 6.3: Statement 2 and 3 modeled in ORM.

Statement 4 - Person with Identification

Statement 4 in Figure 6.1 expresses that every person must have a unique identification, which is either a D-number or a birth number. This statement results in the concepts `Person` which is in a binary relationship with `Identification`. A person's identification will not be divided into a D-number and a birth number, since it will be to close to an implementation. Both identification types contain eleven digits. The first six digits of a birth number is a person's date of birth. In the cases where the identification for a person is a birth number the model

of a person's date of birth will be redundant. An identification should be unique for a person and two people can not have the same identification number. In ORM this is modeled by a mandatory role constraint on the role `has_id` and single internal uniqueness over the roles `has_id` and `is_id_for`. Figure 6.4 shows what this statement is modeled to in ORM.

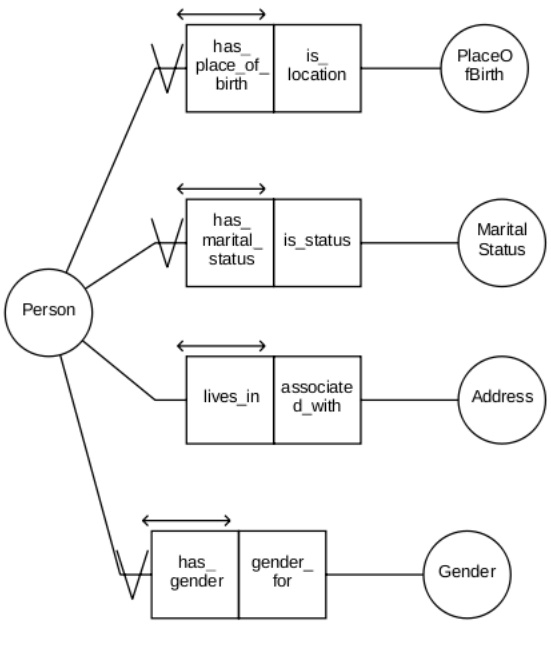


4. Every person must have an identification, which is either a D-number or a birth number

Figure 6.4: Statement 4 modeled in ORM.

Statement 5 - 8 - Additional Personal Information

Statement 5 - 8 in Figure 6.1 expresses that every person must have a place of birth, a gender, a marital status and optionally an address. These statements results in five concepts: `Person`, `PlaceOfBirth`, `Gender`, `MaritalStatus` and `Address`. The concept `Person` will be in a binary relationship with the other concepts. A person should play the roles to place of birth, gender and marital status exactly once. Therefore these roles will have a mandatory role constraint with a single internal uniqueness. While it is optional for a person to have an address, a person may not have more than one address, so the role from `Person` to `Address` has a single internal uniqueness constrain. Figure 6.5 shows what these statements are modeled to in ORM.



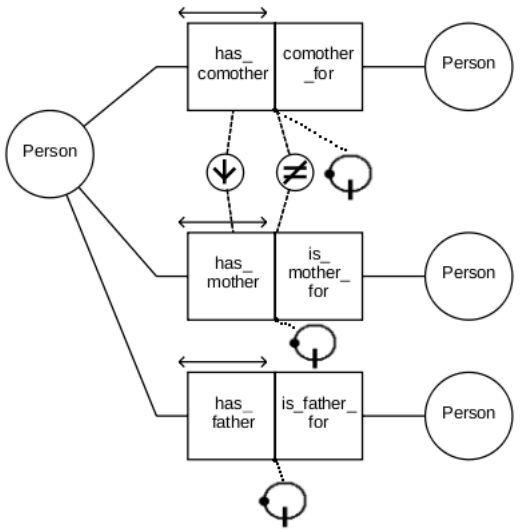
5. Every person must have a place of birth.
6. Every person must have a marital status.
7. Every person must have a gender.
8. Every person may have an address.

Figure 6.5: Statement 5 - 8 modeled in ORM.

Statement 9 - 11 Parenthood

Statement 9 in Figure 6.1 expresses that every person may have a mother and a father, but can not be his/her own parent. Statement 10 in the same figure expresses that a person

with a mother may have a comother, but she can not be the same person. Only the people that already have a mother may have a comother. Statement 11 in the same figure expresses that a person may have one or several children. It is important to mention that the gender of the people in the mother/comother/father relationships will not be taken into account in the model. These statements describe relationships between people and can be modeled with three binary relationships to and from the concept `Person`. The roles are populated with parents and children. For the fatherhood relationship the roles are `has_father` and `is_father_for`. The same structure goes for mother and comother. Although it is optional to have a parent, no person can have more than one father, mother or comother. This is restricted with a single internal uniqueness constraint over the roles `has_father/mother/comother`. Since a person may have several children there are no restrictions on the roles `is_father/mother/comother_for`. The restriction on comother is modeled with a subset constraint from the role `has_comother` to the role `has_mother`. In addition it is necessary to restrict a person to not be mother and comother for the same person. This is done by adding an exclusion constraint between the role pairs `has_mother` - `is_mother_for` and `has_comother` - `is_comother_for`. In addition there is an irreflexive ring constraint over all the role pairs so a person can not be his/her own mother, comother or father. Figure 6.6 shows what these statements are modeled to in ORM.



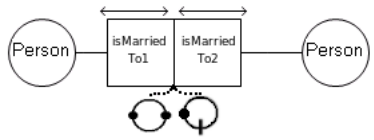
9. Every person may have a mother and a father, but can not be his/her own parent..
10. Every person that has a mother may have a comother, but she can not be the same person.
11. Every person may have one or several children.

Figure 6.6: Statement 9 - 11 modeled in ORM.

Statement 12 - Marriage

Statement 12 in Figure 6.1 expresses that every marriage consists of two different people, and a person can only be married to one other person at the time. The one you are married to are also married to you. In Norway a marriage is gender neutral. This statement describes a relationship between two people. It can be modeled as a binary relationship from and to the concept `Person` with the roles `isMarriedTo1` and `isMarriedTo2`. The first person plays the role `isMarriedTo1` to the second person and the second person plays the role `isMarriedTo2` to the first person. A marriage is optional, but a person can not be married to more than one other person at the time. To express this, both the roles need to have a single internal uniqueness constraint. In addition the role pair needs a symmetric and irreflexive ring constraint. The symmetric ring constraint will ensure that when person1 is married to person2, person2 is also married to person1. The irreflexive ring constraint will prevent a person from being married to

oneself. Figure 6.7 shows what this statement is modeled to in ORM.



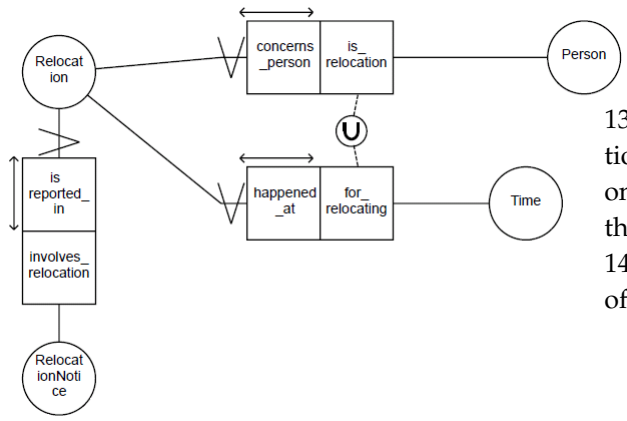
12. Every marriage consist of two different people, and a person can only be married to one other person at the time. The one you are married to are also married to you.

Figure 6.7: Statement 12 modeled in ORM.

Statement 13 - 16 Relocation andS Relocation Notice

Statement 13 in Figure 6.1 expresses that every relocation must be reported in a relocation notice, and applies to exactly one person and one point in time. For a given time and person there should be only one relocation. Statement 14 in the same figure expresses that every relocation notice may involve relocation of people. Statement 15 in the same figure expresses that every relocation notice must have a moving-away address, a moving-to address, a registration time and a time for the relocation. These statements have been divided into two structures, relocation and relocation notice. First is a description of the structure of relocation which is followed by a description of relocation notice.

The model of relocation consists of four concepts: `Relocation`, `RelocationNotice`, `Person` and `Time`. The combination of the person relocating and the time at which the relocation takes place, makes a relocation unique. This means that only one relocation will apply for a given time and person. This is expressed in the model by an external uniqueness constraint between the roles from `Person` and `Time` to `Relocation`. A relocation must be reported in a relocation notice, so the role from `Relocation` to `RelocationNotice` must have a mandatory role constraint and a single internal uniqueness constraint. A relocation notice may involve none or several relocations, so the role from `RelocationNotice` to `Relocation` has no constraints. The roles from `Relocation` to `Time` and `Person` need a mandatory role constaint and a single internal uniqueness since a relocation must play these roles exactly once. Figure 6.8 shows what the statements of relocation are modeled to in ORM.

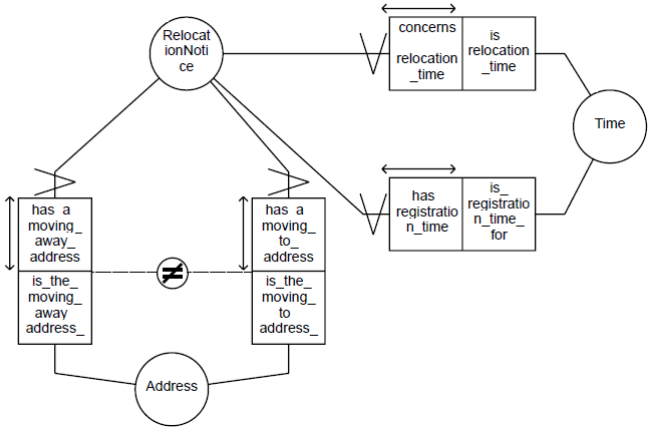


13. Every relocation must be reported in a relocation notice, and applies to exactly one person and one point in time. For a given time and person there should be only one relocation.
14. Every relocation notice may involve relocation of people.

Figure 6.8: Statement 13 and 14 modeled in ORM.

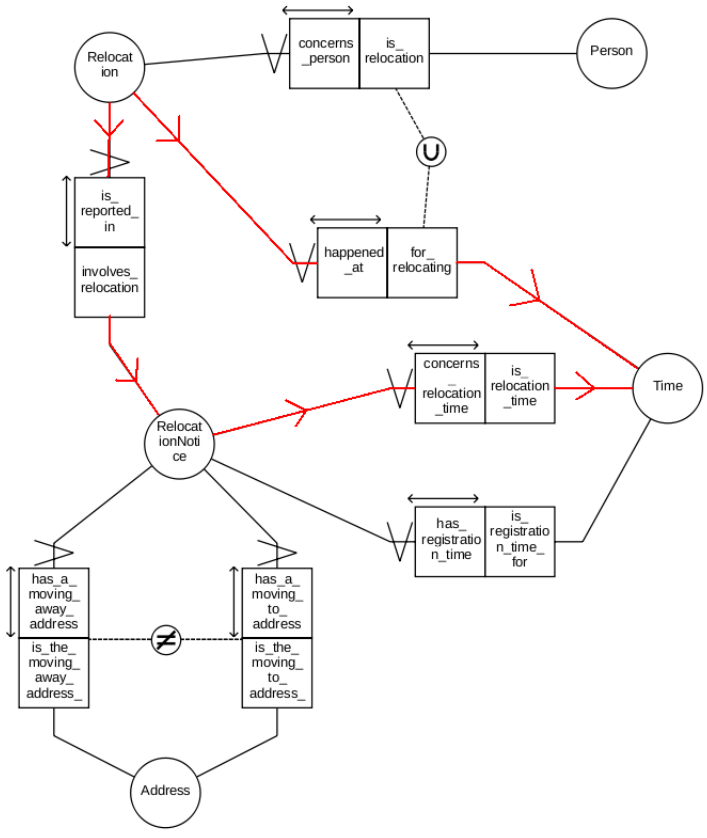
The model of relocation notice consists of three concepts: `RelocationNotice`, `Time` and `Address`. There is a connection from `RelocationNotice` to `Relocation` in Figure 6.8. For every

relocation notice there must be only one moving-away address, one moving-to address, one registration time and one time for the relocation. In order to model that there must be exactly one, the roles from relocation notice to the other concepts get a mandatory role constraint and a single internal uniqueness constraint. There are two roles from `RelocationNotice` to `Address` that concerns moving. It needs to be stated that these addresses are not the same for a relocation notice. This can be done by adding an exclusion constraint between the pair of the roles. Figure 6.9 shows what the statement of relocation notice is modeled to in ORM.



15. Every relocation notice must have a moving-away address, a moving-to address, a registration time and a time for the relocation.

Figure 6.9: Statement 15 modeled in ORM.



16. Every person that is concerned by a relocation notice has the same relocation date.

Figure 6.10: Equivalence-of-path modeled in ORM.

Statement 16 in Figure 6.1 expresses that every person that is concerned by a relocation notice has the same relocation date. In the model this is visible as the model of `Relocation` and the model of `RelocationNotice` are connected and share the same concept `Time`. Since they are intertwined there exist two paths to the same instance of time. One path goes from `Relocation`'s role `happened_at` to `Time`, and the other path goes from `Relocation`'s role `is_reported_in` to `RelocationNotice` and from there to `Time` by following the role `concers_relocation_time`. Both of these paths should lead to the same instance of time. This can be modeled by an Equivalence-Of-Path (EOP) constraint. This constraint on these paths will ensure that the time for the relocation in a relocation notice is the same time as the relocation time in a relocation. Figure 6.10 shows where the equivalence-of-path constraint is on the combined model of relocation and relocation notice.

Statement 17 - 21 - Address

Statement 17 - 21 in Figure 6.1 concerns an address. They express that an address must lie in a country, must belong to an election district, and may lie in a school district, a church parish and a part of town. The statements result in six concepts: `Address`, `Country`, `ElectionDistrict`, `SchoolDistrict`, `ChurchParishes` and `PartOfTown`. The roles that are a non optional for every address, `lies_in_country` and `belongs_to_election_district`, get a mandatory role constraint. All of the roles get a single internal uniqueness constraint because an address can not be in more than one country, election district, school district, church parish or part of town. Figure 6.11 shows what the statements about address are modeled to in ORM.

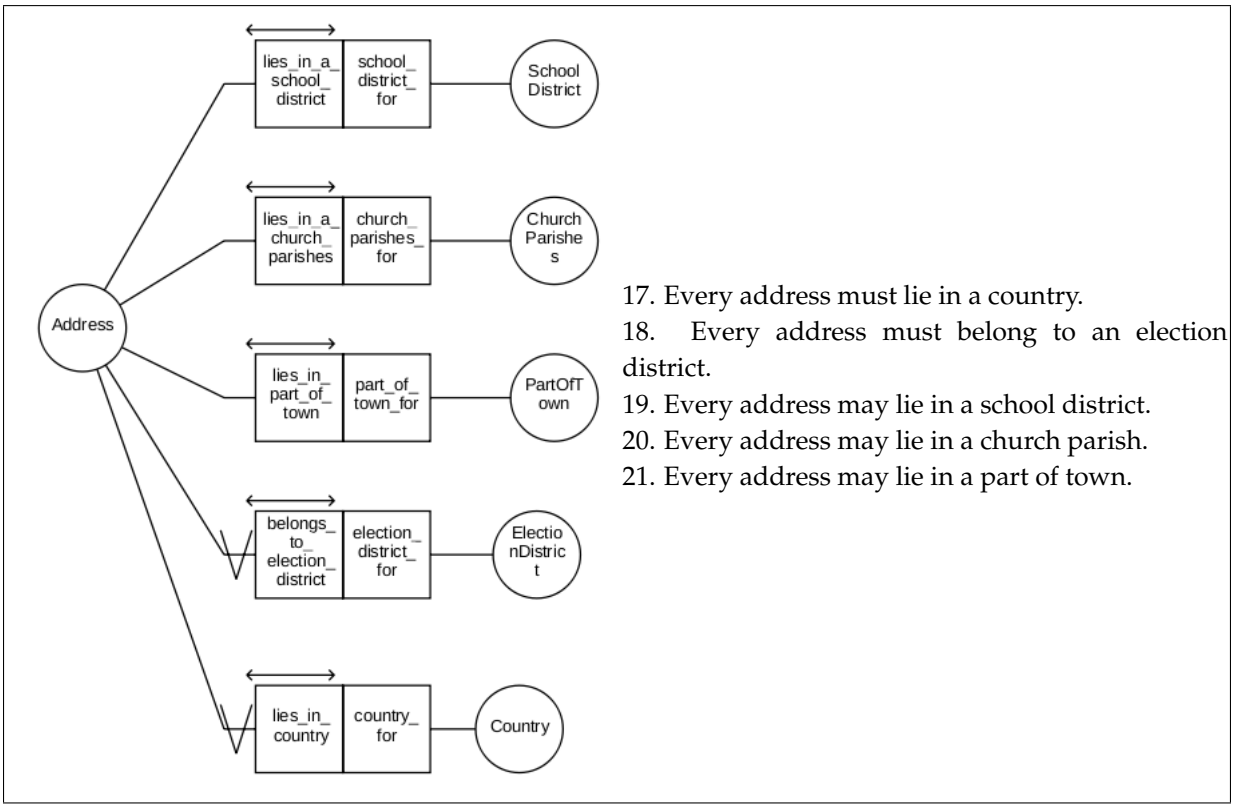


Figure 6.11: Statement 17 - 21 modeled in ORM.

6.4 Mapping from ORM to OWL

In the time of writing there exists no standard, broadly accepted tool that is capable of mapping from ORM to OWL. The closest one is DogmaModeler, which is unavailable and seem inadequate. So in the lack of a proper tool, and for the learning experience, this will be done manually. In addition doing the mapping manually will increase the understanding of the principles behind it.

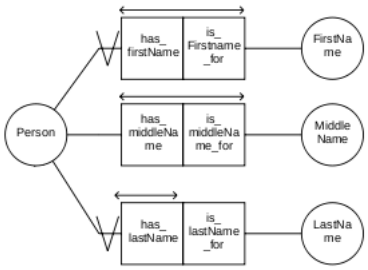
The first idea was to create two models with the same statements by translating a statement in ORM into an equal statement in OWL. After doing some research on the subject of mapping it was revealed that mapping was done by following a set of predetermined rules. There are different mappings, such as Keet's mapping from ORM into description logic (Keet, 2007), Jarrar and Hodrob's mapping from ORM to OWL (Hodrob & Jarrar, 2010) and the mapping from ORM to OWL by Waghi et al. (Wagih et al., 2011). The mapping rules in (Hodrob & Jarrar, 2010; Wagih et al., 2011) has been used as inspiration for this mapping process.

In OWL the names of the properties are made as explanatory as possible, containing the name of the classes it connect, in order to increase the readability of the model. In addition, as a finishing touch, all the classes created must be made disjoint from each other to prevent an individual from being a member of more than one class.

This section will go through the ORM model piece by piece and explain what it is mapped to in OWL. In some of the figures the OWL code is divided in two. In these cases the properties are to the left and classes to the right.

Person with Name

The statements and the ORM model of person with name are seen in Figure 6.2. In this model all the concepts are mapped to classes and relationships to inverse object properties. The object properties links the classes together by domain and range, as described by the mapping rule of facts type in section 5.1.1 on page 37. The mandatory role constraint on the relationship between `Person` and `FirstName` is mapped to `owl:someValuesFrom`, as described in the mapping rule in section 5.1.5 on page 40. The internal uniqueness constraint on the relationship between `Person` and `MiddleName` should be restricted with domain and range as described by the mapping rule in section 5.1.6 on page 41. This is already done in the fact types mapping rule. Last, the mandatory role and internal uniqueness on the relationship between `Person` and `LastName` will be mapped to `owl:qualifiedCardinality` with value 1, as described by the mapping rule in section 5.1.7 on page 41. Figure 6.12 shows the ORM model of person and name and what it is mapped to in OWL.



Object properties:

```
:isFirstNameForPerson rdf:type
owl:ObjectProperty ;
rdfs:domain :FirstName ;
rdfs:range :Person ;
owl:inverseOf :personHasFirstName .

:personHasFirstName rdf:type
owl:ObjectProperty .

:isMiddleNameForPerson rdf:type
owl:ObjectProperty .

:personHasMiddleName rdf:type
owl:ObjectProperty ;
rdfs:range :MiddleName ;
rdfs:domain :Person ;
owl:inverseOf :isMiddleNameForPerson .

:isLastNameForPerson rdf:type
owl:ObjectProperty ;
rdfs:domain :LastName ;
rdfs:range :Person ;
owl:inverseOf :personHasLastName .

:personHasLastName rdf:type
owl:ObjectProperty .
```

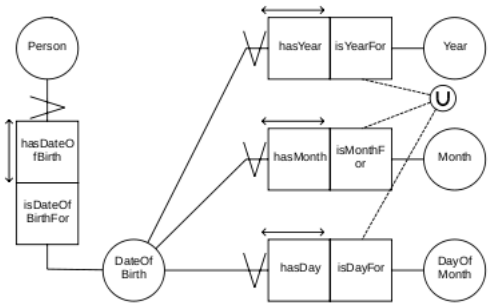
Classes:

```
:FirstName rdf:type owl:Class .

:MiddleName rdf:type owl:Class .

:LastName rdf:type owl:Class .

:Person rdf:type owl:Class ;
rdfs:subClassOf
[ rdf:type owl:Restriction ;
  owl:onProperty :personHasFirstName ;
  owl:someValuesFrom :FirstName ] ,

[ rdf:type owl:Restriction ;
  owl:onProperty :personHasLastName ;
  owl:onClass :LastName ;
  owl:qualifiedCardinality
  "1"^^xsd:nonNegativeInteger ] .
```

Figure 6.12: Mapping of the ORM model person with name to OWL.

Person with Date Of Birth

The statements and the ORM model of person with date of birth are seen in Figure 6.3. In OWL all of the concepts are mapped to classes, resulting in five classes. Each relationship is mapped to inverse object properties, resulting in six properties, according to the mapping rule of fact types described in section 5.1.1 on page 37. The mandatory role and internal uniqueness on the role from `Person` to `DateOfBirth` is mapped to `owl:qualifiedCardinality` with value 1, according to the mapping rule described in section 5.1.7 on page 41. The external uniqueness between the object properties `isYearFor`, `isMonthFor` and `isDayFor` is mapped to a key on the class `DateOfBirth`, by following the mapping rule described in section 5.1.8 on page 42. The key makes sure that the combination of day, month and year is unique for a date of birth. Figure 6.13 shows the ORM model of person with date of birth and what it is mapped to in OWL.



Object properties:

```
:dateOfBirthContainsDay
rdf:type owl:ObjectProperty ;
rdfs:domain :DateOfBirth ;
rdfs:range :DayOfMonth ;

:dayIsPartOfDateOfBirth
rdf:type owl:ObjectProperty .
owl:inverseOf
:dateOfBirthContainsDay .

:dateOfBirthContainsMonth
rdf:type owl:ObjectProperty ;
rdfs:domain :DateOfBirth ;
rdfs:range :Month ;

:monthIsPartOfDateOfBirth
rdf:type owl:ObjectProperty .
owl:inverseOf
:dateOfBirthContainsMonth .

:yearIsPartOfDateOfBirth
rdf:type owl:ObjectProperty ;
rdfs:range :DateOfBirth ;
rdfs:domain :Year ;
owl:inverseOf
:dateofBirthContainsYear .

:dateofBirthContainsYear
rdf:type owl:ObjectProperty .

:isDateOfBirthForPerson
rdf:type owl:ObjectProperty ;
rdfs:domain :DateOfBirth ;
rdfs:range :Person ;
owl:inverseOf
:personHasDateOfBirth .

:personHasDateOfBirth
rdf:type owl:ObjectProperty .
```

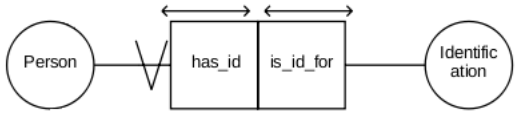
Classes:

```
:DayOfMonth rdf:type owl:Class .

:Month rdf:type owl:Class .

:Year rdf:type owl:Class .

:DateOfBirth rdf:type owl:Class ;
owl:hasKey (
:dateOfBirthContainsDay
:dateOfBirthContainsMonth
:dateOfBirthContainsYear ) .

:Person rdf:type owl:Class ;
rdfs:subClassOf
[ rdf:type owl:Restriction ;
owl:onProperty :personHasDateOfBirth ;
owl:onClass :DateOfBirth ;
owl:qualifiedCardinality
"1"^^xsd:nonNegativeInteger ] .
```

Figure 6.13: Mapping of the ORM model of person with date of birth to OWL.

Person with Identification

The statements and the ORM model of person with identification are seen in Figure 6.4. This model is mapped to a key in OWL by following the mapping rule of a perfect bridge, as described in section 5.1.3 on page 39. Although the concept `Identification` is not a value type, it is still required that a person has a unique identification. The concept `Identification` is mapped to a class. Figure 6.14 shows the ORM model of person and identification and what it is mapped to in OWL.

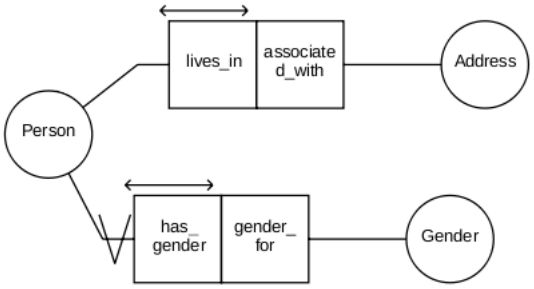



```
:personHasId rdf:type owl:ObjectProperty ;
rdfs:range :Identification ;
rdfs:domain :Person ;
owl:inverseOf :isIdForPerson .

:isIdForPerson rdf:type owl:ObjectProperty .

:Identification rdf:type owl:Class .

:Person rdf:type owl:Class ;
 owl:hasKey ( :personHasId ) .
```

Figure 6.14: Mapping of the ORM model of person with identification to OWL.

Additional Personal Information

The statements and the ORM model of the additional personal information are seen in Figure 6.5. The fact types are mapped to classes and inverse object properties in OWL, according to the mapping rule of fact types described in section 5.1.1 on page 37. The mandatory role constraint and the single internal uniqueness constraint can be combined in OWL. They apply for the relationships from `Person` to `Gender`, `MaritalStatus` and `PlaceOfBirth`. The constraints can be mapped to `owl:qualifiedCardinality` with value `"1"^^xsd:nonNegativeInteger`, according to the mapping rule in section 5.1.7 on page 41. The single internal uniqueness constraint on the relationship from `Person` to `Address` is mapped to `owl:maxQualifiedCardinality` with the value `"1"^^xsd:nonNegativeInteger`, according to the mapping rule described in section 5.1.6 on page 41. This ensures that a person have at most one address. Figure 6.15 shows what the ORM model of gender and address is mapped to in OWL, while Figure 6.16 shows the same for the ORM model of place of birth and marital status.



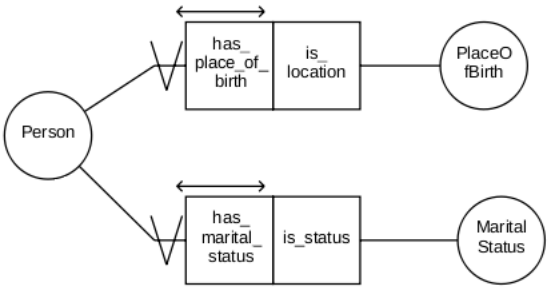
Object properties:

```
:addressAssociatedWithPerson rdf:type
owl:ObjectProperty ;
rdfs:domain :Address ;
rdfs:range :Person ;
owl:inverseOf :personLivesInAddress .

:personLivesInAddress rdf:type
owl:ObjectProperty .

:isGenderForPerson rdf:type
owl:ObjectProperty ;
rdfs:domain :Gender ;
rdfs:range :Person ;
owl:inverseOf :personHasGender .

:personHasGender rdf:type
owl:ObjectProperty .
```

Classes:

```
:Address rdf:type owl:Class .

:Gender rdf:type owl:Class .

:Person rdf:type owl:Class ;
rdfs:subClassOf
[ rdf:type owl:Restriction ;
owl:onProperty :personHasGender ;
owl:onClass :Gender ;
owl:qualifiedCardinality
"1"^^xsd:nonNegativeInteger ] ,

[ rdf:type owl:Restriction ;
owl:onProperty :personLivesInAddress ;
owl:onClass :Address ;
owl:maxQualifiedCardinality
"1"^^xsd:nonNegativeInteger ] .
```

Figure 6.15: Mapping of the ORM model of person with gender and address to OWL.



Object properties:

```
:isMaritalStatusForPerson rdf:type
owl:ObjectProperty ;
rdfs:domain :MaritalStatus ;
rdfs:range :Person ;
owl:inverseOf :personHasMaritalStatus .

:personHasMaritalStatus rdf:type
owl:ObjectProperty .

:isPlaceOfBirthForPerson rdf:type
owl:ObjectProperty ;
rdfs:range :Person ;
rdfs:domain :PlaceOfBirth ;
owl:inverseOf :personHasPlaceOfBirth .

:personHasPlaceOfBirth rdf:type
owl:ObjectProperty .
```

Classes:

```
:MaritalStatus rdf:type owl:Class .

:PlaceOfBirth rdf:type owl:Class .

:Person rdf:type owl:Class ;
rdfs:subClassOf
[ rdf:type owl:Restriction ;
owl:onProperty :personHasPlaceOfBirth ;
owl:onClass :PlaceOfBirth ;
owl:qualifiedCardinality
"1"^^xsd:nonNegativeInteger ] ,

[ rdf:type owl:Restriction ;
owl:onProperty :personHasMaritalStatus ;
owl:onClass :MaritalStatus ;
owl:qualifiedCardinality
"1"^^xsd:nonNegativeInteger ] .
```

Figure 6.16: Mapping of the ORM model of person with place of birth and marital status to OWL.

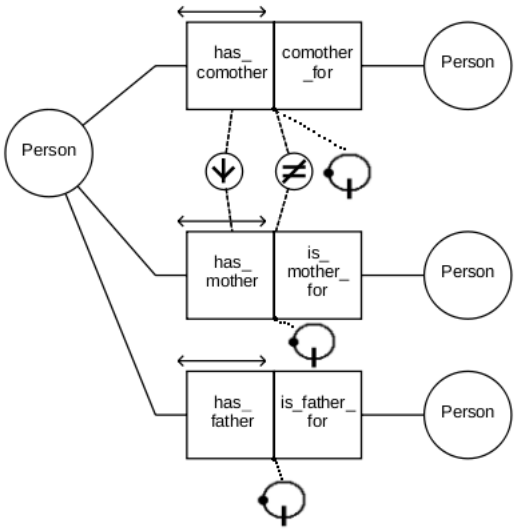
Parenthood

The statements and the ORM model of parenthood are seen in Figure 6.6. The concepts in this parenthood are mapped to classes and the roles to inverse object properties, according to the mapping rule of fact types described in section 5.1.1 on page 37. The object properties `personHasFather/Mother/CoMother` are mapped to `owl:IrreflexiveProperty` by following the mapping rule of ring constraints in section 5.1.22 on page 53. This prevents a person from being his/her own father, mother or comother. In addition the object properties `personIsFather/Mother/CoMotherFor` should be mapped to both `owl:IrreflexiveProperty` and `owl:InverseFunctionalProperty`. This to prevent a person from being his/her own parent and to ensure that a person has no more than one father, mother or comother.

The single internal uniqueness on the roles `hasFather/Mother/CoMother` is mapped to `owl:maxQualifiedCardinality` with value 1, according to the mapping rule of internal uniqueness described in section 5.1.6 on page 41. To be able to construct that a person must have a mother before the person can have a comother, two additional class need to be made in OWL. These classes catch the set of individuals that have a mother or a comother, according to the mapping rule of subsets describes in section 5.1.20 on page 51. These classes are called `PersonWithMother` and `PersonWithCoMother`. The class `PersonWithMother` should have as an equivalent class the object properties `personHasMother` with the cardinality restriction

owl:someValuesFrom, to ensure that a person of this type has a mother. In addition the class should be a subclass of class Person. On the other hand, the class PersonWithCoMother should have as an equivalent class the object properties personHasCoMother with the cardinality restriction owl:someValuesFrom, to ensure that a person has a comother. It should be made a subclass of PersonWithMother. Last the object property personHasCoMother should have as domain PersonWithCoMother and range Person.

In order to avoid that a person can be mother and comother to the same individual the object properties personIsMother/CoMotherFor should be disjoint, according to the mapping rule of exclusion constraint in section 5.1.10 on page 44. Figure 6.17 shows how the ORM model of parenthood and what it is mapped to in OWL.



Object properties:

```
:personIsFatherFor rdf:type
owl:InverseFunctionalProperty ,
owl:IrreflexiveProperty ,
owl:ObjectProperty ;
rdfs:domain :Person ;
rdfs:range :Person ;
owl:inverseOf :personHasFather .

:personHasFather rdf:type
owl:IrreflexiveProperty ,
owl:ObjectProperty .

:personIsMotherFor rdf:type
owl:InverseFunctionalProperty ,
owl:IrreflexiveProperty ,
owl:ObjectProperty ;
owl:inverseOf :personHasMother .

:personHasMother rdf:type
owl:IrreflexiveProperty ,
owl:ObjectProperty ;
rdfs:domain :Person ;
rdfs:range :Person ;
rdfs:domain :PersonWithCoMother ;
rdfs:range :PersonWithMother .

:personIsCoMotherFor rdf:type
owl:InverseFunctionalProperty ,
owl:IrreflexiveProperty ,
owl:ObjectProperty ;
rdfs:domain :Person ;
rdfs:range :PersonWithCoMother ;
owl:inverseOf :personHasCoMother .

:personHasCoMother rdf:type
owl:IrreflexiveProperty ,
owl:ObjectProperty .
```

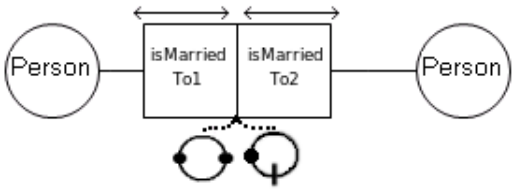
Classes:

```
:Person rdf:type owl:Class ;
rdfs:subClassOf
[ rdf:type owl:Restriction ;
owl:onProperty :personHasFather ;
owl:onClass :Person ;
owl:maxQualifiedCardinality
"1"^^xsd:nonNegativeInteger ] ,

[ rdf:type owl:Restriction ;
owl:onProperty :personHasCoMother ;
owl:onClass :Person ;
owl:maxQualifiedCardinality
"1"^^xsd:nonNegativeInteger ] ,

[ rdf:type owl:Restriction ;
owl:onProperty :personHasMother ;
owl:onClass :Person ;
owl:maxQualifiedCardinality
"1"^^xsd:nonNegativeInteger ] .

:PersonWithMother rdf:type owl:Class ;
owl:equivalentClass
[ rdf:type owl:Restriction ;
owl:onProperty :personHasMother ;
owl:someValuesFrom :Person ] ;
rdfs:subClassOf :Person .

:PersonWithCoMother rdf:type owl:Class ;
owl:equivalentClass
[ rdf:type owl:Restriction ;
owl:onProperty :personHasCoMother ;
owl:someValuesFrom :Person ] ;
rdfs:subClassOf :PersonWithMother .
```

Figure 6.17: Mapping of the ORM model of parenthood to OWL.

Marriage

The statements and the ORM model of marriage are seen in Figure 6.7. Since this construction is between two instances of the same concept, it can be made into one object property in OWL, called `isMarriedToPerson`. The object property should be made irreflexive and symmetric by following the mapping rule of ring constraints described in section 5.1.22 on page 53. The ring constraints prevents a person from being married to oneself and to ensure that the person one is married to also is married to you. The internal uniqueness is mapped to `maxQualifiedCardinality` with the value `"1"^^xsd:nonNegativeInteger`, according to the mapping rule described in section 5.1.6 on page 41. The cardinality constraint ensures that a person can only be married once (at the time). Figure 6.18 shows the ORM model of marriage and what it is mapped to in OWL.



Object property:

```
:isMarriedToPerson rdf:type
owl:ObjectProperty ,
owl:IrreflexiveProperty ,
owl:SymmetricProperty ;
rdfs:domain :Person ;
rdfs:range :Person .
```

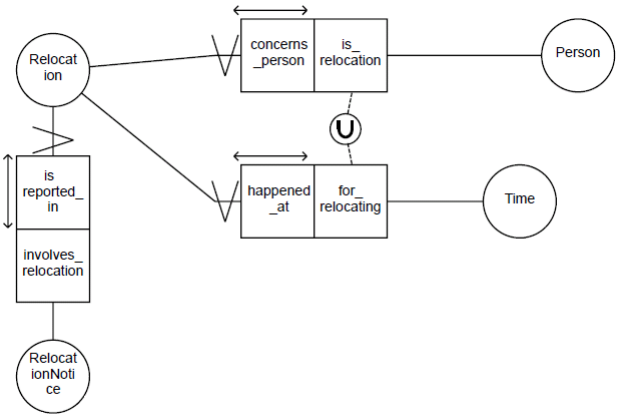
Class:

```
:Person rdf:type owl:Class ;
rdfs:subClassOf
[ rdf:type owl:Restriction ;
owl:onProperty :isMarriedToPerson ;
owl:onClass :Person ;
owl:maxQualifiedCardinality
"1"^^xsd:nonNegativeInteger ] .
```

Figure 6.18: Mapping of the ORM model of a marriage to OWL.

Relocation

The statements and the ORM model of relocation are seen in Figure 6.8. In OWL all the concepts are mapped to classes and all the roles to inverse object properties, according to the rule of mapping fact types described in section 5.1.1 on page 37. The mandatory role constraint and the internal uniqueness constraint are combined in OWL. They appear together on the role from `Relocation` to `RelocationNotice` and are mapped to `owl:qualifiedCardinality` with the value `"1"^^xsd:nonNegativeInteger`, according to the mapping rule described in section 5.1.7 on page 41. The external uniqueness between the object properties `relocationConcernsPerson` and `relocationHappenedAtTime` is mapped to a key, according to the mapping rule described in section 5.1.8 on page 42. Figure 6.19 shows what this construction looks like in ORM and what it is mapped to in OWL.



Object properties:

```
:personIsRelocating rdf:type
owl:ObjectProperty ;
rdfs:domain :Person ;
rdfs:range :Relocation ;
owl:inverseOf
:relocationConcernsPerson .

:relocationConcernsPerson
rdf:type owl:ObjectProperty .

:relocationHappendAtTime rdf:type
owl:ObjectProperty .

:timeIsReloactionTimeForRelocation
rdf:type owl:ObjectProperty ;
rdfs:range :Relocation ;
rdfs:domain :Time ;
owl:inverseOf :relocationHappendAtTime .

:relocationIsReportedInRelocationNotice
rdf:type owl:ObjectProperty ;
rdfs:domain :Relocation ;
rdfs:range :RelocationNotice ;
owl:inverseOf
:relocationNoticeInvolvesRelocation .

:relocationNoticeInvolvesRelocation
rdf:type owl:ObjectProperty .
```

Classes:

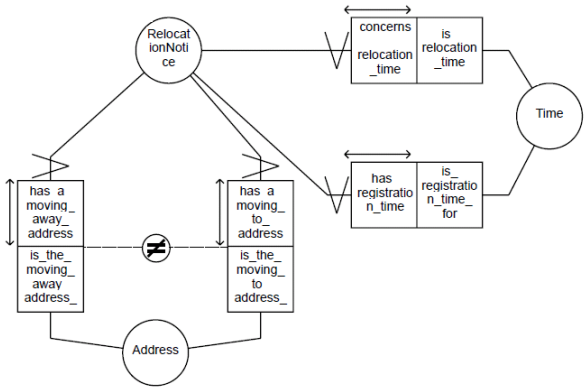
```
:Person rdf:type owl:Class .

:Time rdf:type owl:Class .

:Relocation rdf:type owl:Class ;
rdfs:subClassOf
[ rdf:type owl:Restriction ;
owl:onProperty :relocationHappendAtTime ;
owl:onClass :Time ;
owl:qualifiedCardinality
"1"^^xsd:nonNegativeInteger ] ,

[ rdf:type owl:Restriction ;
owl:onProperty :relocationConcernsPerson ;
owl:onClass :Person ;
owl:qualifiedCardinality
"1"^^xsd:nonNegativeInteger ] ,

[ rdf:type owl:Restriction ;
owl:onProperty
:relocationIsReportedInRelocationNotice ;
owl:onClass :RelocationNotice ;
owl:qualifiedCardinality
"1"^^xsd:nonNegativeInteger ] ;

owl:hasKey
( :relocationConcernsPerson
:relocationHappendAtTime ) .

:RelocationNotice rdf:type owl:Class .
```

Figure 6.19: Mapping of the ORM model for relocation to OWL.

Relocation Notice

The statements and the ORM model of relocation notice are seen in Figure 6.9. In OWL the concepts are mapped to classes and the roles to inverse object properties, according to the rule of mapping fact types described in section 5.1.1 on page 37. The mandatory role constraint and

the single internal uniqueness constraint that appear together on all the roles from `Relocation`, are combined to one constraint in OWL. This constraint is `owl:qualifiedConstraint` with value `"1"^^xsd:nonNegativeInteger`, according to the mapping rule described in section 5.1.7 on page 41. The exclusion constraint between the pair of roles from `RelocationNotice` to `Address` is mapped by making the pair of object properties disjoint with one another, according to the mapping rule described in section 5.1.10 on page 44. This property ensures that the addresses in the two object properties are not the same for one relocation notice. Figure 6.20 shows how the structure of relocation notice looks like in ORM and what it is mapped to in OWL.



Object properties:

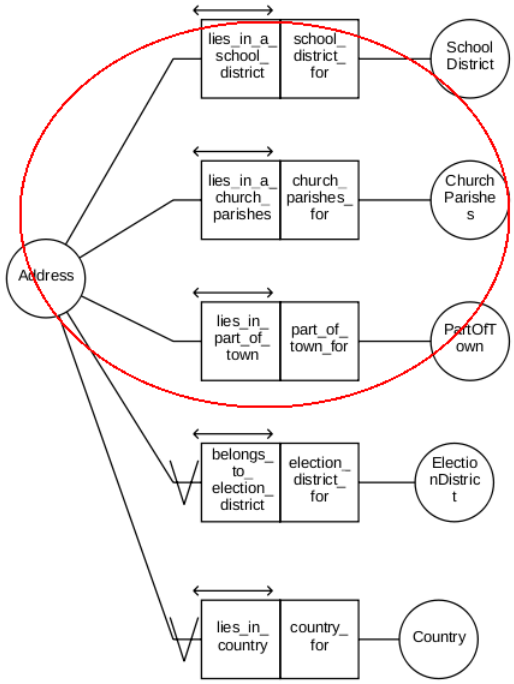
```
:isMovingAwayAddressForRelocationNotice
rdf:type owl:ObjectProperty ;
rdfs:domain :Address ;
rdfs:range :RelocationNotice ;
owl:propertyDisjointWith
:isMovingToAddressForRelocationNotice ;
owl:inverseOf
:relocationNoticeHasMovingAwayAddress .

:relocationNoticeHasMovingAwayAddress
rdf:type owl:ObjectProperty .

:isMovingToAddressForRelocationNotice
rdf:type owl:ObjectProperty ;
rdfs:domain :Address ;
rdfs:range :RelocationNotice ;
owl:inverseOf
:relocationNoticeHasMovingToAddress .

:relocationNoticeHasMovingToAddress
rdf:type owl:ObjectProperty .

:relocationNoticeConcernsRelocationTime
rdf:type owl:ObjectProperty .

:timeIsRelocationTimeForRelocationNotice
rdf:type owl:ObjectProperty ;
rdfs:range :RelocationNotice ;
rdfs:domain :Time ;
owl:inverseOf
:relocationNoticeConcernsRelocationTime .

:relocationNoticeHasRegistrationTime
rdf:type owl:ObjectProperty  .

:timeIsRegistrationTimeForRelocationNotice
rdf:type owl:ObjectProperty ;
rdfs:range :RelocationNotice ;
rdfs:domain :Time ;
owl:inverseOf
:relocationNoticeHasRegistrationTime .
```

Classes:

```
:Address rdf:type owl:Class .

:RelocationNotice rdf:type owl:Class ;
rdfs:subClassOf
[ rdf:type owl:Restriction ;
owl:onProperty
:relocationNoticeConcernsRelocationTime ;
owl:onClass :Time ;
owl:qualifiedCardinality
"1"^^xsd:nonNegativeInteger ] ,

[ rdf:type owl:Restriction ;
owl:onProperty
:relocationNoticeHasRegistrationTime ;
owl:onClass :Time ;
owl:qualifiedCardinality
"1"^^xsd:nonNegativeInteger ] ,

[ rdf:type owl:Restriction ;
owl:onProperty
:relocationNoticeHasMovingAwayAddress ;
owl:onClass :Address ;
owl:qualifiedCardinality
"1"^^xsd:nonNegativeInteger ] ,

[ rdf:type owl:Restriction ;
owl:onProperty
:relocationNoticeHasMovingToAddress ;
owl:onClass :Address ;
owl:qualifiedCardinality
"1"^^xsd:nonNegativeInteger ] .

:Time rdf:type owl:Class .
```

Figure 6.20: Mapping of the ORM model of relocation notice to OWL.

Figure 6.10 shows where the equivalent of path is in the ORM model. This constraint is not possible to construct in the OWL language. The reason why and consequences will be discussed in Chapter 8.



Object properties:

```
:addressLiesInSchoolDistrict rdf:type
owl:ObjectProperty .

:schoolDistrictForAddress rdf:type
owl:ObjectProperty ;
rdfs:range :Address ;
rdfs:domain :SchoolDistrict ;
owl:inverseOf :addressLiesInSchoolDistrict .

:addressLiesInChurchParishes rdf:type
owl:ObjectProperty .

:churchParishesForAddress rdf:type
owl:ObjectProperty ;
rdfs:range :Address ;
rdfs:domain :ChurchParishes ;
owl:inverseOf :addressLiesInChurchParishes .

:addressLiesInPartOfTown rdf:type
owl:ObjectProperty ;
rdfs:domain :Address ;
rdfs:range :PartOfTown ;
owl:inverseOf :partOfTownForAddress .

:partOfTownForAddress rdf:type
owl:ObjectProperty .
```

Classes:

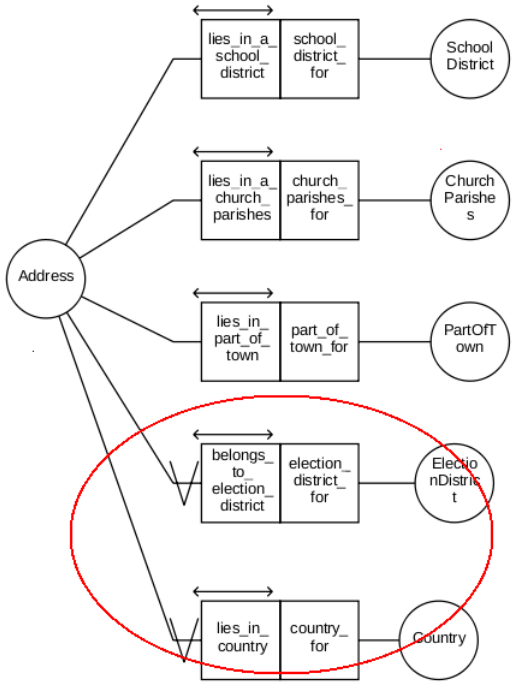
```
:SchoolDistrict rdf:type owl:Class .

:ChurchParishes rdf:type owl:Class .

:PartOfTown rdf:type owl:Class .

:Address rdf:type owl:Class ;
rdfs:subClassOf
[ rdf:type owl:Restriction ;
owl:onProperty :addressLiesInSchoolDistrict ;
owl:onClass :SchoolDistrict ;
owl:maxQualifiedCardinality
"1"^^xsd:nonNegativeInteger ] ,

[ rdf:type owl:Restriction ;
owl:onProperty :addressLiesInChurchParishes ;
owl:onClass :ChurchParishes ;
owl:maxQualifiedCardinality
"1"^^xsd:nonNegativeInteger ] ,

[ rdf:type owl:Restriction ;
owl:onProperty :addressLiesInPartOfTown ;
owl:onClass :PartOfTown ;
owl:maxQualifiedCardinality
"1"^^xsd:nonNegativeInteger ] ,
```

Figure 6.21: Mapping of the first part of the ORM model of address to OWL.

Address

The statements and the ORM model of address are seen in Figure 6.11. In OWL the roles are mapped to inverse object properties and the concepts are mapped to classes, according to the

mapping rule of fact types described in section 5.1.1 on page 37. The mandatory role constraint and the internal uniqueness constraint that appear together on the roles from `Address` to `ElectionDistrict` and `Country`, can be combined to one constraint in OWL. This constraint is `owl:qualifiedCardinality` with value `"1"^^xsd:nonNegativeInteger`, according to the mapping rule described in section 5.1.7 on page 41. The single internal uniqueness on the role from `Address` to the remaining three concepts is mapped to `owl:maxQualifiedCardinality` with value `"1"^^xsd:nonNegativeInteger`, according to the mapping rule described in section 5.1.6 on page 41. This constraint expresses that there has to be at most one value for the restricted property. Because the OWL code takes up a lot of space the mapping has been divided into two figures. Figure 6.21 shows the entire ORM model of the address and what the first part is mapped to in OWL, while Figure 6.22 shows the entire ORM model of address and what the second part is mapped to in OWL.



Object properties:

```
:addressLiesInCountry rdf:type
owl:ObjectProperty .

:countryForAddress rdf:type
owl:ObjectProperty ;
rdfs:range :Address ;
rdfs:domain :Country ;
owl:inverseOf :addressLiesInCountry .

:addressBelongsToElectionDistrict rdf:type
owl:ObjectProperty .

:electionDistrictForAddress rdf:type
owl:ObjectProperty ;
rdfs:range :Address ;
rdfs:domain :ElectionDistrict ;
owl:inverseOf :addressBelongsToElectionDistrict .
```

Classes:

```
:Country rdf:type owl:Class .

:ElectionDistrict rdf:type owl:Class .

:Address rdf:type owl:Class ;
rdfs:subClassOf
[ rdf:type owl:Restriction ;
owl:onProperty :addressLiesInCountry ;
owl:onClass :Country ;
owl:qualifiedCardinality
"1"^^xsd:nonNegativeInteger ] ,

[ rdf:type owl:Restriction ;
owl:onProperty :addressBelongsToElectionDistrict ;
owl:onClass :ElectionDistrict ;
owl:qualifiedCardinality
"1"^^xsd:nonNegativeInteger ] .
```

Figure 6.22: Mapping of the second part of the ORM model of address to OWL.

Chapter 7

Connecting Data from the Database with an OWL Model using D2RQ

This chapter gives an explanation to the mapping performed from the database to OWL by using the tool D2RQ. First is a section that describes the mapping from the ORM model to the database schema. Next is a section that explains which changes were made to the D2RQ mapping file to make it map to the OWL model. Last is a section that shows the result of the mapping with D2RQ.

7.1 Mapping from the ORM Model to a Database Schema

Before the ORM model could be mapped to a database schema, it was necessary to create a reference to all the concepts in the model, so the tables in the relational database could have unique identifiers. The names of the columns and tables in the database are generated by the Relational Mapping Procedure (Halpin et al., 2008). The names of tables and columns were changed in order to increase readability. stORM has a feature that allows the user to decide the names of tables and column before the mapping to the database is performed.

Figure 7.1 shows the ORM structure of `Address`. The concepts that are connected to concept `Address` become columns in the table `Address` with foreign keys to their own tables. Figure 7.2 shows the table `Address` with a row describing the address `Forge Cottage`.

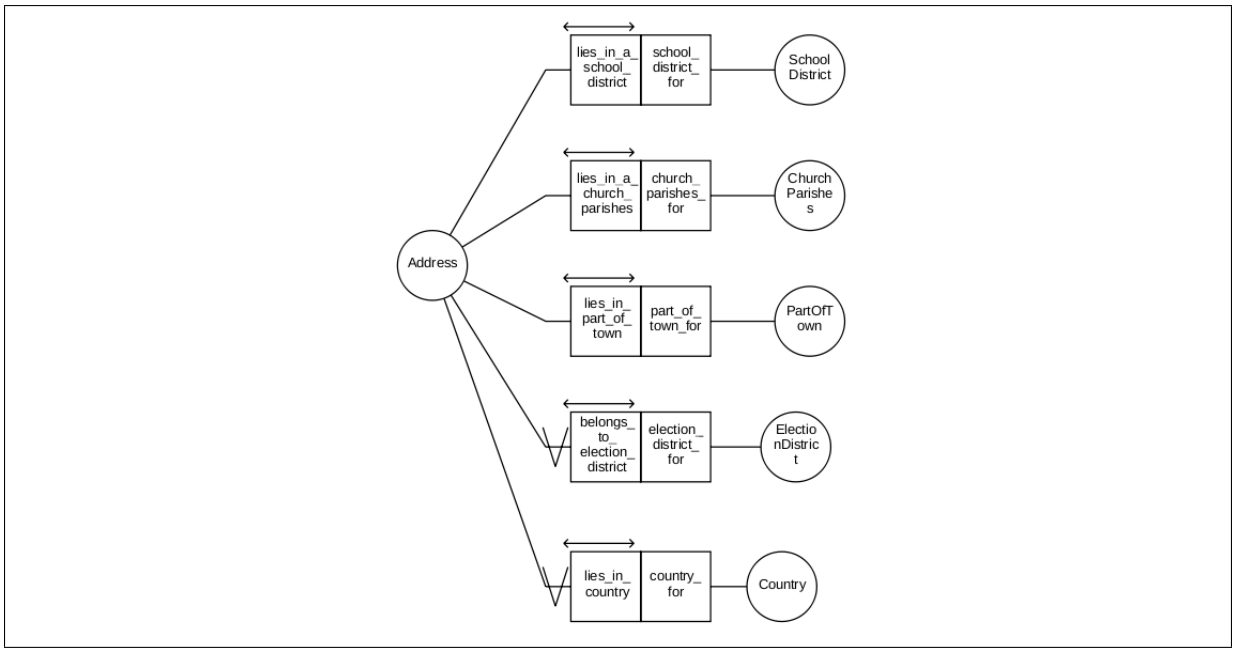


Figure 7.1: The ORM structure of `Address`.

addr	country	election_district	church_parishes	part_of_town	school_district
Forge Cottage	England	Greendale	North Yorkshire		Greendale

Figure 7.2: The database table `Address` with address `Forge Cottage`.

The database schema created by the mapping is seen in Appendix B, section B.1. It contains the definition of 24 tables, seen in Figure 7.3. See Appendix B; section B.2 for an overview of the tables' structure. There are three tables more than there are concepts in the ORM model. They are `Marriage`, `Person_FirstName` and `Person_MiddleName`. The tables `Person_FirstName` and `Person_MiddleName` are created because fact types with an internal uniqueness that spans both roles are turned into tables of their own. These tables describe the many-to-many relationship between the concepts. The table `Marriage` is created because fact types with two internal uniquenesses over the binary relationship are turned into tables of their own. The table describes the one-to-on relationship between instances of the same concept.

The tables are filled with a minimum of data that will serve for demonstration purposes. All of the data in the database is purely fictional and any similarities to people in the real world are a coincidence. After the database schema was created it was necessary to add checks to ensure that the symmetry and irreflexivity are maintained, since stORM does not support this.

7.2 The Mapping

The tool chosen to do the mapping from the database to an OWL model is the tool D2RQ, the engine-part. The tool is a platform which consists of the tools D2R server, D2RQ engine and a prototype extension D2RQ/Update and D2R Update Server (D2R Update, 2011). D2RQ

Address	MiddleName
ChurchParishes	Month
Country	PartOfTown
DateOfBirth	Person
DayOfMonth	Person_FirstName
ElectionDistrict	Person_MiddleName
FirstName	PlaceOfBirth
Gender	Relocation
Identification	RelocationNotice
LastName	SchoolDistrict
MaritalStatus	Time
Marriage	Year

Figure 7.3: The tables created by the Relational Mapping Procedure.

fulfilled most of the tool requirements. It maps the database to an OWL model, keeps the data in the database, and can query the resulting ontology with SPARQL. In addition the tool is available, free of charge and well documented. It has a rich API for generating the map from a database to either a RDF vocabulary or an OWL model.

D2RQ's automatic mapping process is performed by executing a command in a terminal. The result is by default a mapping file that describes a mapping to a RDF vocabulary. In order to make the mapping file map to an OWL model instead one must manually change the file. In our case we want to map the database to the OWL model created by the mapping from ORM. In order to merge the data from the database with the OWL model, and make the data a part of the model, the two need to share the same vocabulary. The automatically generated vocabulary in the mapping file is replaced by the vocabulary for the OWL model, so that the value of the mapping classes and properties becomes the OWL model's classes and properties. This way the database gets mapped to the OWL model instead of the RDF vocabulary. The OWL model's vocabulary, the T-box of the model, is available at *http://sws.ifi.uio.no/vocab/dsf#*.

The ClassMaps in the mapping file describes how the tables in the database are mapped to classes in the OWL model. The value of the property `d2rq:class` is the OWL class that the tables are mapped to. The PropertyBridges describes how the columns in the database are mapped to properties. The value of the property `d2rq:property` is the OWL property that the value of the column is mapped to. The mapping file consists of 24 ClassMaps, one for each table in the database and one PropertyBridge for each of the columns in the tables.

The ClassMaps that map the tables `Marriage`, `Person_FirstName` and `Person_MiddleName`, are removed. Since these tables hold the relationship between concepts that already have tables of their own, the values in them are mapped to properties in the OWL model that describes these relationships. The values in the `Marriage` table are mapped to the property `isMarriedTo`, while the values of `Person_FirstName/MiddleName` are mapped to the properties `personHasFirstName/MiddleName` and `isFirst/MiddleNameForPerson`. These properties connect married people and people with their first/middlename the same way as the ORM model does.

To increase the browsing possibilities and to make the model easier to understand some adjustments are made to the PropertyBridges. In the PropertyBridges that tells the label of a ClassMap, `d2rq:pattern` is replaced by `d2rq:column`. The value of this property is set to be the table's referencing column. In addition a datatype is added by `d2rq:datatype`. This makes the label of the OWL classes the literal value of their mapped table's referencing

column which resides in the database. In the PropertyBridges that map to a foreign key in the database, `d2rq:column` is replaced by a join expressed with `d2rq:join` and a URL expressed by `d2rq:uriPattern`. The URL points to the class that the foreign key links to. This will increase readability since the value of the properties will be a named individual of an OWL class instead of only a literal value. For those PropertyBridges that refers to an instance within the same ClassMap, like `personHasMother/Father/CoMother`, `personIsMother/Father/Comother` and `personIsMarriedTo`, it is necessary to use an alias. The alias is used with a join and a URL to refer to the other instance of the same class. The entire mapping file can be view in the Appendix in Chapter D on page 141.

The table `Address` in Figure 7.2 is the basis for the mapping to the class `Address` in the OWL model. Figure 7.4 shows what the address `Forge Cottage` looks like when it is exposed through D2R Server.

property	hasValue	isValueOf
dsf:addressAssociatedWithPerson	db:Person/04057598577	-
dsf:addressAssociatedWithPerson	db:Person/15015499872	-
dsf:addressAssociatedWithPerson	db:Person/25085255175	-
dsf:addressBelongsToElectionDistrict	db:ElectionDistrict/Greendale	-
dsf:addressLiesInChurchParishes	db:ChurchParishes/North_Yorkshire	-
dsf:addressLiesInCountry	db:Country/England	-
dsf:addressLiesInSchoolDistrict	db:SchoolDistrict/Greendale	-
rdf:type	dsf:Address	-
rdfs:label	"Forge Cottage"^^xsd:string	-
dsf:churchParishesForAddress	-	db:ChurchParishes/North_Yorkshire
dsf:countryForAddress	-	db:Country/England
dsf:electionDistrictForAddress	-	db:ElectionDistrict/Greendale
dsf:personLivesInAddress	-	db:Person/04057598577
dsf:personLivesInAddress	-	db:Person/15015499872
dsf:personLivesInAddress	-	db:Person/25085255175
dsf:relocationNoticeHasMovingAwayAddress	-	db:RelocationNotice/3451
dsf:schoolDistrictForAddress	-	db:SchoolDistrict/Greendale

Figure 7.4: The individual `Forge Cottage` exposed by D2R Server.

7.3 The Result

The resulting OWL model is populated with all the data from the database. All the values are labeled correctly and the individuals are member of the correct classes. After performing reasoning over the model the people with mothers and comothers become members of the correct subclasses of `Person`. Reasoning can be done in Protégé with the reasoner Pellet.

The mapped database is made available online by the tool D2R Server (Bizer & Cyganiak, 2007) at the Web address `http://sws.ifi.uio.no/project/dsf/`. Here you can also download the populated OWL model. The server can provide the user with a RDF or HTML view of the database which allows the user to browse the content. In addition the user can perform SPARQL queries against the database through a SPARQL Endpoint. The mapped database does not contain the classes `PersonWithMother` and `PersonWithCoMother` since no values from the database are mapped to these classes. Those classes get populated during reasoning, and the server shows a read-only view of the data from the database with their class membership and relationships to each other. Figure 7.5 shows the classes mapped to by the mapping file.

As seen in Figure 7.4, the D2R Server shows all the connection an individual have to all other classes, both as subject and object of triples. It is possible to extract the same information from the database, but it will require a long query.

Figure 7.5: The classes created by D2RQ.

Chapter 8

Discussion

This chapter will start with a presentation of the lessons learned through the mapping processes from ORM to OWL and from database to OWL. It is followed by a section about how related work differs from the work done in this thesis.

8.1 Discoveries Made During the Mapping Process from ORM to OWL

The mapping from ORM to OWL was performed to investigate what similarities and differences there are between ORM and OWL as languages for creating information models. The aim was to find out if OWL is capable of creating the same structures and constraints as ORM. This section will give an overview over what was learned during the mapping process.

8.1.1 Similarities

The mapping process revealed that there are many similarities between the two languages, in terms of what they are able to express.

Concepts and Classes

Concepts in ORM and classes in OWL are quite similar to each other. Although a class is a type and is used to describe a set of individuals that share some common characteristics, and a concept is used to represent something in the world, they both represent and describe a part of the domain they model. A concept `Address` in ORM plays certain roles, meaning it has a relationship to certain other concepts, and a class `Address` in OWL is a subclass of different property restrictions which means that it describes a set of individuals that have certain relationships to other classes.

Roles and Properties

Roles in ORM and properties in OWL are also quite similar. They both describe and represent the relationship between concepts/classes. The difference is that roles are always connected to a concept, while properties exist independently of classes. The properties' independence is easily dealt with by restricting the domain and range of the properties to only one class. As long as the classes are disjoint the restrictions are powerful enough to ensure that an individual from another class can not use the property. The classes need to be disjoint from each other so that an individual in the class `Address` is not allowed to also be inferred to be an individual in class `Person`.

Datatypes and Perfect Bridges

Datatype properties in OWL are the closest one get to the structure bridge in ORM. A datatype property has the option of being a functional property, which restricts an individual of a class to have no more than one value for the property. For a datatype property to be a satisfying translation of a perfect bridge, it needs to be inverse functional to ensure that two different individuals of a class do not have the same value. The lack of an inverse functional restriction makes is necessary to put keys on all classes created from concepts with perfect bridges.

Keys

Unique identifiers are essential in databases because they separate instances and make them distinct. ORM uses perfect bridges to create unique identifiers in models that will be realized as relational databases. In OWL keys can uniquely identify individuals in classes. In addition they are necessary for the construction of the external uniqueness constraint in ORM, which uniquely identifies a concept based on several roles from different concepts.

Hierarchy and Inheritance

ORM has a concept hierarchy which allows a subtype to inherit the roles played by a supertype in addition to playing roles of its own. OWL has a class hierarchy and a property hierarchy. The class hierarchy is similar to the concept hierarchy in ORM considering that the subclasses inherit the properties from the superclasses. The property hierarchy is similar to a subset on a role pair in ORM considering that a subject and an object connected by a subproperty automatically also gets connected by the superproperty. An example is the property `eats` which has a subproperty `indulgence`. If an individual from class `Person` called "Emanuel" is in an `indulgence` relationship with an individual from class `Food` called "Pasta", then "Emanuel" is also in an `eats` relationship with "Pasta". In addition a subproperty inherits the domain and range from the superproperty. ORM can have subsets on single roles, where only one part of the role pair is affected, as seen in the ORM model of the Norwegian National Register with the roles `hasMother` and `hasCoMother`. To construct this in OWL it is necessary to create extra classes to capture those individuals that have a certain property.

Constraints

There is also a similarity in the expression of value and cardinality constraints in the two languages. The constraints in OWL are capable of creating the same restrictions as the ones in ORM, as long as the properties are additionally restricted with domain and range. The value or cardinality restrictions alone are not sufficiently restrictive as they do not prevent individuals from other classes from also using the properties.

8.1.2 Language Constructions

There are no equivalent language construction in OWL for the ORM construction of n-ary relationships and the constraints multi-role frequency and external frequency.

The mapping done in this thesis focuses on the ability to express the same statement in the two languages without information loss. By this standard, the lack of these constructions and constraints in OWL does not present a problem. OWL can construct equivalent statements with other constructions and constraints. How they are restructured is described in details in their mapping rule in Chapter 5. The mapping rule of ternary fact types, is described in details in Section 5.1.2 on page 38. The mapping rule of multi-role frequency constraint is described in detail in Section 5.1.18 on page 49 and the mapping rule of external frequency constraint is seen in Section 5.1.19 on page 50.

Such restructuring unfortunately comes with the price of reduced readability. The statements in an OWL construction are far less readable than they are in the corresponding ORM construction. This reveals a limitation in OWL. That a statement is difficult to read and requires extra effort to be understood is a sign that the statement has been poorly captured by the model. Poor readability in a model is one aspect that may indicate that the modeling language is not well suited for the task.

A language's suitability as well as its capability to model information is some of the criteria for being a successful information modeling language. The fact that OWL lacks equal language constructions will make it more difficult to create models that capture the intended meaning of statements. Capability alone may not be enough if the model is difficult to create, and difficult to read once it is created.

8.1.3 Equivalence Of Path can not be Expressed in OWL

Equivalence-of-path (EOP) constraint is the only constraint in ORM that is impossible to create in OWL. This is a constraint that is used much in information systems that reuses resources. In OWL it is possible to create paths with property chains, but it is not possible to make a property chain an equivalent property or subproperty of another property (Blace, 2009). In other words one can make a path from `Relocation` to `Time`, as seen in Figure 6.10, and another path from `Relocation` to `RelocationNotice` and then to `Time`, but there is no way of stating that these paths lead to the same individual, the same time, in the class `Time`. It is possible to express EOP by the use of individuals and keys to make an instance unique, but when individuals are added the model is no longer a general model and therefore not comparable to an ORM model. The lack of EOP in the OWL language restricts the language's abilities to handle systems with

reservations or logging, which is a drawback considering that many information systems deal with these kinds of activities.

8.1.4 Reasoning

One major difference between the languages is reasoning. OWL's ability to reason over its models makes it possible to infer new knowledge based on the knowledge that already exists. In addition reasoning can check if the model is consistent.

All modeling tools for OWL have reasoning, while there are no tools for ORM that performs reasoning and infers new knowledge. ORM has the necessary constructions to be reasoned over considering that it has a formal definition in description logic. Jarrar shows this when he maps from ORM to SHOIN/OWL description logic in (Jarrar, 2007), as well as Keet when she maps ORM to the description logic language DLRifd in (Keet, 2007). The lack of reasoning in ORM is due to a lack of tool support, not language capability. OWL has consistency checks on both the syntax and the semantics of its models, while ORM only has consistency checks on the syntax.

The advantage with inferring new knowledge is that it makes implicit knowledge available to the user. When information is inferred and added to the model it makes more information available since the relationships between the information is now explicitly stated. In addition when information is explicitly stated, more information will appear in a user's result set when querying.

On the other hand, reasoning might lead to unintentional information being added to the model if one does not restrict it properly. It is essential to remember to restrict domain and range on properties if they are to connect only two classes, and to make classes that are meant to contain different individuals disjoint. If not the reasoner might draw unintentional conclusions and infer unwanted information.

Consistency checking on the semantic in a model is practical since it can alert the user that inconsistent information has been added to the model. It helps preventing incorrect knowledge since the reasoner will state the OWL model as inconsistent if it detects information which directly violates the restriction put on the model.

OWL's consistency checking alerts when restrictions have been violated, but not when they have yet to be fulfilled. This is most visible on property restriction put on classes, and is an effect of the Open World Assumption which concludes that information that is not found is unknown rather than false.

8.1.5 Open World Assumption

In the view of the Semantic Web an Open World Assumption (OWA) is practical because "Anyone can say Anything about Anything". Information can be added to the Semantic Web by anyone at anytime and the reasoner has to draw conclusions based on the available data. There are more information in the world about topics than what is available to the reasoner in a Semantic Web model. Therefore the reasoner does not conclude that information is false simply because it is lacking. After all, it might be added tomorrow. By this standard it is

an advantage to add all available information to the model, even if it is not complete. The lacking information can simply be added at a later time when it is known. In the meantime this information is available in the model, and can be reasoned over which might lead to the discovery of more information.

The reasoner will not state that information does not exist simply because it is lacking. Let us say you add a person to one of the OWL models created in this thesis, but you do not add the name, because you do not know it. The property restriction on class `Person` states that every person added to the model must have exactly one `LastName`. The reasoner knows this, but concludes that the name simply is not known yet. This assumption allows you to add incomplete information to the model, since the reasoner does not state that the model is inconsistent. Depending of what information is lacking this can be a disadvantage. If information that is supposed to be mandatory is lacking it allows violation of the rules on which the model it created. Let us say that a relocation notice is to be registered in the model and it lacks the destination address. This violates the rule that every relocation notice must have a moving-to address. In addition it is very inconvenient for the users of the model to not know where people have moved to. How will one get hold of the people that have moved? What is the purpose of adding a relocation notice if the main part, the new address, is lacking? The extra work required to find the missing address will frustrate the people operating the model.

Another construction that allows incomplete information to be added is the subclass construction. In the mapping from ORM to OWL subclasses are used to express a subset between mother and comother, forcing a person to have a mother before he/she can have a comother. This constraint does not need to be met for the model to be consistent. It is possible to add a person to the model and give it a comother without having a mother, because the reasoner knows that the person is suppose to have a mother, she is merely unknown. The reasoner does not complain because the mother might be added later. This allows violation of a rule on which the information model is built.

As another example, imagine an information model with professional drivers, like lorry drivers, taxi drivers and limousine drivers. To make sure that all of the people who drive vehicles for a living have a license, the class `PeopleWithDriversLicense` is a superclass of all the different drivers. With an Open World Assumption it is possible to add an individual to the class `TaxiDriver` without stating that the person has a driver's license. The person may have a driver's license that no one knows about, but as long as it is unknown he/she should not be allowed to work as a taxi driver. There is no way for the model to express whether or not this person is qualified.

Another consequence of the OWA is the restrictions on what can be reasoned. A reasoner is not capable of inferring that individuals lack a certain property and thereby collect them in one class, unless the individuals are forced to not have the property. If people have to have either a car or a bike, but can not have both, they will be forced to be in one class or the other. In this case it is possible for a reasoner to infer that some people do not have a car and make them members of the `PersonWithoutCar` class. Although it is not possible for a reasoner to collect individuals that lack a certain property, it is possible to query for all individuals that does. The information is still possible to extract from a model.

A tool to Remove Open World Assumption

If the OWL reasoners of tomorrow are better at checking for inconsistency and thereby capable of forcing restrictions in the model to be met, it will improve OWL's capability as a language for information modeling.

The ability to violate the rules of the model by adding incomplete information is a concern of ongoing research. Clark & Parsia is working on a new reasoner called Pellet Integrity Constraint Validator (ICV) (Pellet ICV, 2011). This tool aims to close up the world and introduce a weak Unique Name Assumption by checking for inconsistencies before it infers new knowledge. Clark & Parsia are producing this reasoner as a respond to feedback and user experience.

8.1.6 Non Unique Name Assumption

The Non Unique Name Assumption (NUNA) has the effect that two individuals with different names are not necessarily different. In the view of the Semantic Web where "Anyone can Call Anything Anything", information about the same topic can be added through many different individuals by many different people. Let us say you add the individual *Leonardo Da Vinci* to a model on the Semantic Web, and you state that he was an artist. Another person has earlier added information about the individual *Leonardo Davinci* and stated that he was an engineer. They are both referring to the same person, but with different names due to a spelling error. These two individuals can, because of NUNA, be stated as the same individual without loosing any information. NUNA makes redundant and ambiguous data easier to handle.

Since the reasoner does not separate classes and individuals on their names it is necessary to state that they are the same or different in order to avoid unintentional information to be inferred. In the OWL models made in this thesis, all the classes have different names and are semantically different from each other. It is necessary to explicitly state this by making all the classes disjoint from another so the reasoner does not infer that an individual belongs to more than one class.

In contrast to OWL, ORM uses the Unique Name Assumption in which different names indicate difference. In an ORM model it is not necessary to explicitly state that concepts are different because everything is implicitly different. The exception is subtypes, which are the same type as their supertype.

8.1.7 The OWL model Mapped from ORM

ORM has a methodology for creating models. It is described in the Conceptual Schema Design Procedure's seven step (Halpin et al., 2008) and in Skagestein's rules of thumb (Skagestein, 1996). By following the methodology one will end up with a precise and accurate model that captures the Universe Of Discourse. OWL does not have a methodology, which makes it is difficult to provide a clear answer to what kind of structures are best to capture the intended meaning. There are several ways to express the same statement, and one is not more correct than the other. Which structure to use is a matter of personal preferences.

The OWL model mapped from ORM captures almost all of the statements in Figure 6.1 meaning the model is correct. This is a truth with modifications considering the Open World Assumption allows the rules to be violated. It is possible to express the same statements in another way than they are currently expressed, by using fewer classes and more datatypes. It will not be more correct then the current structure, only different. Considering that the model captures the statements, more or less correct, one can not say that another structure will be better since there is no standard structure in the first place.

8.2 Mapping from Database to the OWL Model with D2RQ

The mapping from a database to an ontology was performed to see if OWL can support and underpin an information system. It is very interesting to see how data from a database can be combined with an OWL model, since there are no tools for ORM that supports this combination.

The combination of the metadata in an OWL model and the data values are practical since it allows changes in the metadata to immediately affect the data. If one changes the constraints in an OWL model to be more restrictive the reasoner can state that it is inconsistent since there are more properties than allowed, which will require the data to be restructured. If one changes the constraints to be less restrictive one allows more data to be added to the model than previously. In addition the combination of data and metadata makes it easier to interchange information between information systems. The metadata allows the data values to be incorporated automatically in another system, since the statement of relationships between the data values are independent of their storage. Therefore it is desirable to make data in databases available in an OWL model, and this can be achieved with mapping.

The populated OWL model contains all of the data from the database. It is possible to reason over the model and the combination of model and data is easy to share since the relationship between the data is independent of its storage. The model is sharable through the Web and it exploits the advantages of the database when it comes to performance on query execution and storage. Another advantage with this solution is that when records in the database are added or deleted the data in the model changes as well. This is not the case with models that contains the extracted data, since the data set only will contain the data in the database at time of extraction.

In our case the database contains some tables for which the OWL model does not have corresponding classes. These are `Person_FirstName`, `Person_MiddleName` and `Marriage`. In the OWL model the values of these tables are incorporated into object properties. Marriage is mapped to a symmetric object property between two people. The OWL model has two subclasses of `Person` called `PersonWithMother` and `PersonWithCoMother`. `PersonWithCoMother` is in addition a subclass of `PersonWithMother`. These classes are not mapped from the database, but they get populated when the reasoner infers that some people match the membership criteria.

The mapped database can be put on the Web by the tool D2R Server. This tool provides the user with a HTML or RDF view of the data in the database, in addition to a SPARQL endpoint where the user can query the content. It is important to mention that D2RQ is not able to create or delete tables in the database, so changes made to the structure of the database requires a new mapping process to be conducted.

The extension prototype of D2RQ, D2RQ/Update and D2R Update Server (D2R Update, 2011) are capable of updating the data in the database by deleting and inserting data values from the model layer. There was no time to test these tools, so they are not a part of this thesis. In order to use these tools another mapping was required by another mapping language created especially for the tools and the task. An interesting aspect of these tools is that the ontology must abide the constraints from the database in addition to the constraints of the model. Since the D2R Update Server inserts and deletes data in the database the constraints from the database is transferred to the model (Hert, Reif, & Gall, 2010). This means that the constraints in the database control what can be inserted and deleted (Hert et al., 2010). The database constraints are the basis for the consistency checking, so the tool takes into account primary/foreign keys and non-nullable columns. This has the effect that the Open World Assumption is closed up, but only when using D2R Update Server.

Databases combined with ontologies are cutting-edge technology. There is still a way to go before it is possible to make changes to the structure of the database from a model layer with today's tools. If one extracts the data from the database and creates a populated ontology, it is possible to make changes to the constraints and structure of the model, and have it affect the consistency of it. The drawback is that one looses the strength of the database when it comes to query performance and storage, and changes made to the data in the database are not reflected in the ontology.

Because of the time limit on the thesis there was no time to test the capabilities of the query rewriting part of D2RQ. In addition the rewriting algorithm is a black box and is poorly documented. It would be interesting to compare the result of SPARQL queries towards the ontology, with SQL queries towards the database. In addition it would be interesting to compare the speed of the execution for the different queries to see which one performs best on simple and complex queries.

8.3 Related Work

To our knowledge there are only two relevant articles that concerns mapping from an ORM model to an OWL 2 model. These are Hodrob and Jarrar's article "ORM to OWL 2 DL Mapping" (Hodrob & Jarrar, 2010) and the article "Mapping Object Role Modeling 2 Schemes to OWL2 Ontologies" by Wagih et al. (Wagih et al., 2011). In addition there is Keet's article "Mapping the Object-Role Modeling language ORM2 into Description Logic language DLRifd" (Keet, 2007) that maps from ORM to Description Logic. All of these articles give a detailed description of the mapping rules used for the mapping process. Hodrob and Jarrar maps from ORM to SHOIN and then to OWL, while Wagih et al. maps directly from ORM to OWL. Wagih et al. uses Hodrob and Jarrar's article as an inspiration and have some mapping rules that differ from their work.

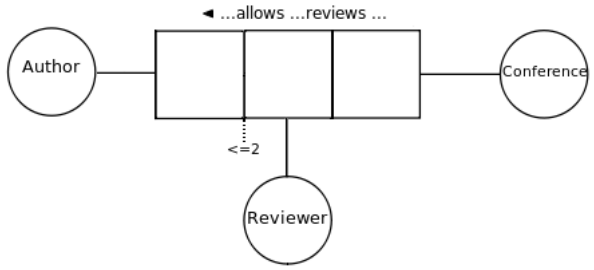
Some of the mapping rules presented in this thesis differs from the rules presented by Wagih et al. and Hodrob and Jarrar. It is necessary to mention that the mapping rule of internal multi-role frequency constraint, presented by Wagih et al., is incorrect. In addition the claims made by Wagih et al. that external uniqueness constraint, external frequency constraint and n-ary fact types are impossible to map to OWL are also incorrect. Hodrob and Jarrar seems, as far as we can see, to be correct in their mapping, but unfortunately there are many mapping rules they do not include like partition constraint, exclusive-or constraint, perfect bridges and external uniqueness.

This section will start by taking a look at the mapping rule of internal multi-role frequency, presented by Wagih et al. It is followed by the mapping rule of external uniqueness constraint and external frequency constraint that was stated as impossible to map from ORM to OWL, which we mapped by using other constructions and constraints.

8.3.1 Disagreement on Mapping Rule

Internal Multi-Role Frequency Constraint

Internal multi-role frequency constraint is an ORM constraint that Hodrob and Jarrar did not consider in their article, since SHOIN does not support it. Wagih et al. argue in (Wagih et al., 2011) that the constraint can be expressed using equivalence and cardinality constraints as seen in the bottom left part of Figure 8.1. We argue that the cardinality constraint on class `Conference` is not sufficient to express this constraint and that one should not use equivalence to map it. In their solution a conference is defined by who is "allowed", which we do not think is a well defined definition. In addition the cardinality constraint `allows` with range individuals that are both `Reviewer and Author` does not capture the constraint. This constraint allows the value to be max two individuals that are both reviewer and author, for every conference. The ORM model that it the basis for this mapping in (Wagih et al., 2011), seen in the top of Figure 8.1, expresses that there is allowed max two of the same pair where one value is from `Author` and one value is from `Reviewer` for the entire set. This means that a conference will only have one person that is both reviewer and author, since each combination of author, reviewer and conference need to be unique. We think that the internal multi-role frequency constraint needs to be expressed by adding an extra class `ReviewingAuthor` that combines the two roles touches by the constraint. This new class will have a `owl:maxQualifiedCardinality` constraint with value 2 on the object property that connects it to `Conference`. In addition `ReviewingAuthor` is made unique by the combination of the object properties that connects it to `Author` and `Reviewer`. This construction expresses that for every combination of author and reviewer there can be maximum 2 different conferences.



Written in Manchester syntax:

```
 Class: Conference
EquivalentTo: allows max 2
(Reviewer and Author)
```

Written in Turtle syntax:

```
:ReviewingAuthor rdf:type owl:Class ;
rdfs:subClassOf
[ rdf:type owl:Restriction ;
  owl:onProperty :reviews ;
  owl:onClass :Conference ;
  owl:maxQualifiedCardinality
  "2"^^xsd:nonNegativeInteger ] ;
owl:hasKey ( :hasAuthor, :hasReviewer ) .
```

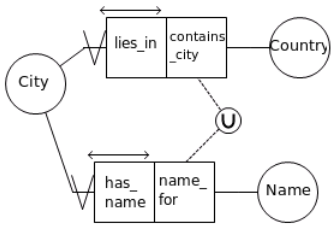
Figure 8.1: Differences on the mapping of internal multi-role frequency constraint.

8.3.2 Constructions and Constraint Stated as Impossible to Map to OWL

External Uniqueness Constraint

Wagih et al. claims that it is not possible to map the external uniqueness constraint in ORM to an OWL construction. We propose to construct the constraint by using a key to create a uniqueness for a class based on the properties that connects it to the different classes. Wagih et al. says in (Wagih et al., 2011) that OWL can not create this constraint, but shows that Robinson et al. in (Robinson, Henricksen, & Indulska, 2007) suggests the solution of wrapping properties into either an objectification or a new class. Figure 8.2 shows our solution for the mapping of an external uniqueness constraint.

Wagih et. al claims it is not possible to map an external uniqueness constraint.



Our solution:

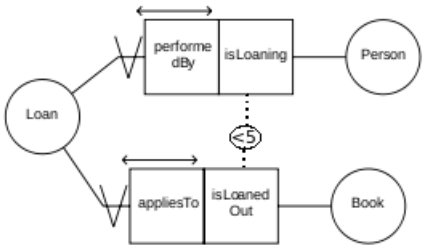
```
:City rdf:type owl:Class ;
owl:hasKey (:liesInCountry, :hasName) .
```

Figure 8.2: Differences on the mapping of an external uniqueness constraint.

External Frequency Constraint

Another ORM constraint that Wagih et al. claims is impossible to map to OWL is the external frequency constraint. We state that it is possible to construct the same statement expressed by this constraint in OWL by adding an extra class. The extra class combines the classes touched by the external frequency constraint and. It is made unique by the object properties that connect to them. In addition the extra class expresses the frequency as an internal frequency constraint. Figure 8.3 shows only the extra combination class since the entire example takes up a lot of space. See section 5.1.19 on page 50 for the mapping rule that describes in details how this constraint is mapped to OWL.

Wagih et. al claims it is not possible to map an external frequency constraint.




```
                    ...
                    ...
:LoanCase rdf:type owl:Class ;
rdfs:subClassOf
[ rdf:type owl:Restriction ;
  owl:onProperty :loanCaseConstituteLoan ;
  owl:onClass :Loan ;
  owl:maxQualifiedCardinality
  "5"^^xsd:nonNegativeInteger ] ;
owl:hasKey (:loanCasePerformedByPerson,
            :loanCaseAppliesToBook) .
                   ...
                   ...
```

Figure 8.3: Differences on the mapping of an external frequency constraint.

Chapter 9

Conclusion

The mapping from ORM to OWL reveals some significant differences in the languages' capability and suitability to create information models.

The structures of the languages proved to be fairly similar. Classes and concepts both represent a small part of the world they model, and properties and roles describe the relationship between them. In addition almost all of the constructions and constraints in ORM are possible to construct in OWL which shows that OWL can handle the structure of a well defined information model.

The mapping process also revealed that OWL has some features which make information modeling complicated. OWL uses the Non Unique Name Assumption which requires us to take some precautions when modeling. Considering that two classes contains different individuals, meaning that the classes are different, need to be explicitly stated to prevent an individual from being a member of more than one class.

The feature in OWL that prevents it from being a successful modeling language, as seen from a database modeler's viewpoint, is the Open World Assumption. OWL permits incomplete knowledge in a model since it might be added later. This has the consequence that restrictions put on a class can be violated, since the reasoner concludes that the lacking information simply is not known yet. By violating restrictions on classes the model violates the rules on which it is built. A way to deal with this limitation may be to use new reasoners. Clark & Parsia are developing a new reasoner, Pellet Integrity Constraint Validator, which aims to close the Open World Assumption by performing consistency checks before it infers new knowledge. It also aims to introduce a weak Unique Name Assumption.

The Open World Assumption and the Non Unique Name Assumption are very practical considering the Semantic Web. The Non Unique Name Assumption makes redundant and ambiguous data on the Web easier to handle. If two individuals or two classes are found to have the same semantic meaning they can be combined without any information loss. The Open World Assumption allows information to be added to a model bit by bit as it becomes known. This is possible because information that is not found in a model is stated as unknown rather than false, so the model is still consistent even if some information is lacking.

ORM has some constraints and constructions that can not be expressed with a similar language structure in OWL. These are n-ary relationships, internal multi-role frequency constraints and

external frequency constraints. The statements they express can be constructed in OWL with some restructuring. The fact that restructuring is required to model certain statements in OWL may make it more difficult for modelers to create a model that correctly captures the intended meaning. In addition the restructuring leads to reduced readability which makes the extraction of the statement more difficult than it is in ORM. This makes OWL less suitable as a modeling language, although not any less capable.

One constraint in ORM was found to be impossible to construct in OWL and that is the Equivalence-Of-Path constraint. This constraint appears frequently when resources are reused, meaning there are two paths leading to the same instance of a concept. OWL's lack of this constraint restricts its functionality to not model areas or activities that deals with logging or ticketing, since they often reuse resources.

OWL's ability to support and underpin an information system was investigated by researching on tools that combine the data from databases with an OWL model. OWL outperforms ORM in this aspect, considering that there are no tools for ORM which allows it to connect data to its models. For OWL several tools exist which extract and expose the data in different ways. The ones we have looked at either extract the data from the database and create a populated ontology or they create a bridge to it. None of them supports adding of new constraints or tables to the database.

The tool chosen to perform the mapping from the database to an OWL model is D2RQ, which creates a bridge to the data in the database. The automatic mapping generates a mapping file which needs to be modified in order to change the target from a RDF vocabulary to an OWL model. Some of the tables in the database are not mapped to classes in the OWL model, but object properties. All the data from the database were mapped to individuals of the correct OWL classes. The mapped database can be put on the Web by D2R Server and the OWL model's vocabulary can be put online. This gives the advantage of easy sharing at the same time as one utilizes the database's strength in query performance and storage.

To summarize, ORM performs better than OWL in information modeling considering that the restrictions on classes in OWL do not need to be met due to the Open World Assumption, and that ORM can express statements more clearly. OWL has a reduced readability compared to ORM on complex constructions and it does not have the ability to create the Equivalence-of-Path constraint. On the other hand OWL outperforms ORM on the tool support for connecting the model to the data in a database. D2RQ allows the data from the database to be connected with the model. The combined model is easy to share considering the data and the relationships between them are independent of storage. In addition the model can be reasoned over.

9.1 Further Work

Many of the problematic aspects discovered in this thesis can be solved with more advanced tools. In future work it will be interesting to take a close look at Pellet Integrity Constraint Validator to see how it can remove the Open World Assumption and introduce a weak Unique Name Assumption. It can make OWL more capable of creating models for information systems. We are curious of how it works and which constraints that will be met by using it versus another reasoner.

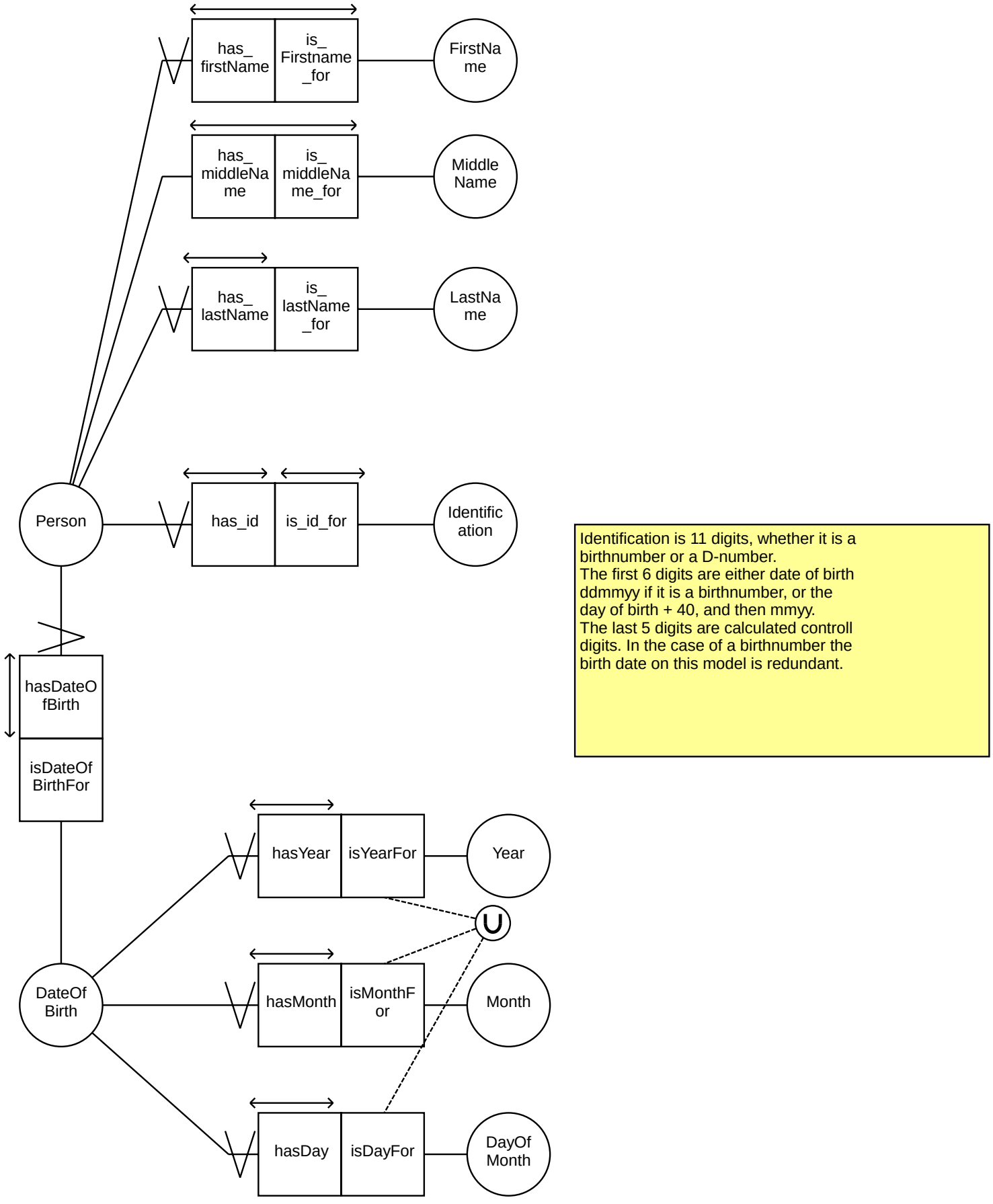
Another tool that should be further investigated is D2RQ. It should be tested with a real

size database to see how it handles the amount of data and check if there are any scalability problems. A real size database should also be used for testing the query capabilities of D2RQ. D2RQ rewrites SPARQL queries to SQL, and it would be interesting to investigate how it performs compared to ordinary SQL queries executed against the database. The result set and speed on the query execution should be compared to see if there are any significant differences or problems.

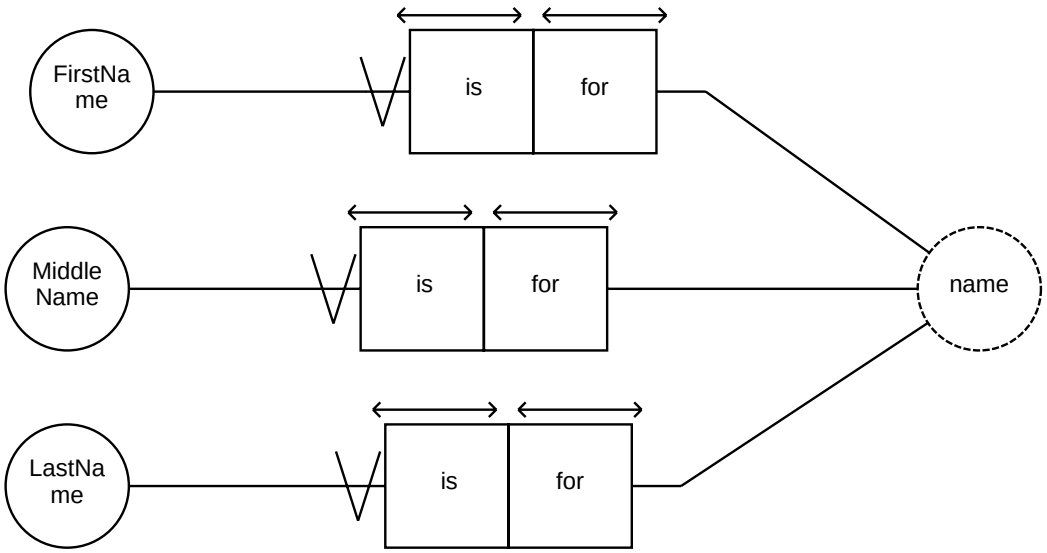
In addition the D2R Update Server should be looked at more closely. It is interesting to find out how it works and check if it is capable of removing and inserting data from different tables at once. Last it would be interesting to generally investigate if there are any works in progress on mapping tool to solve the task of changing the structures of databases from the model layer.

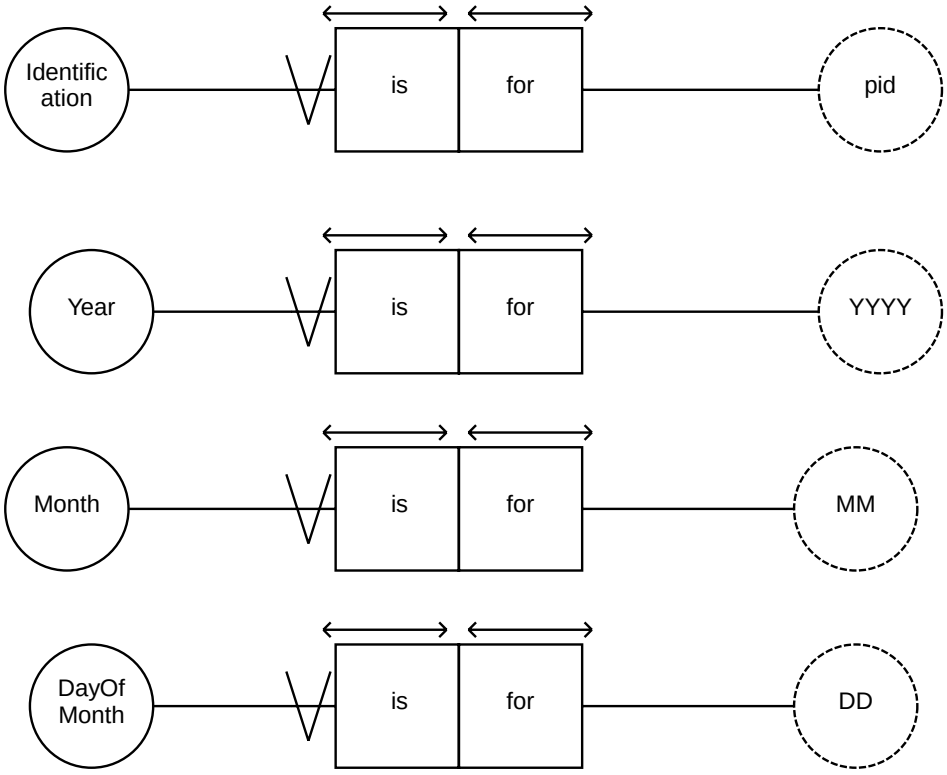
Appendix A

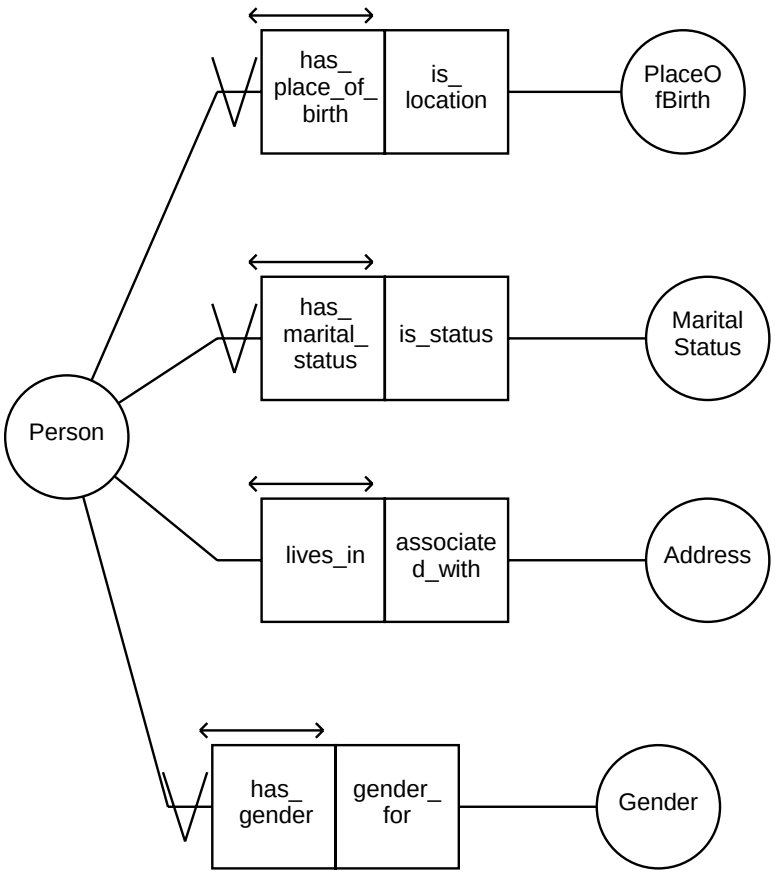
The Complete ORM Model of the Norwegian National Register

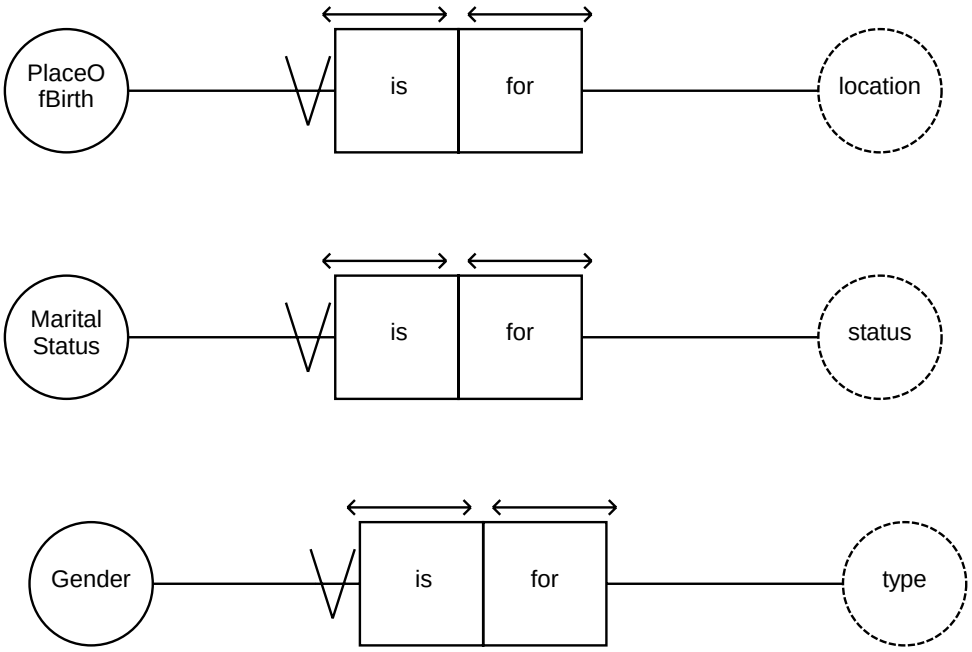


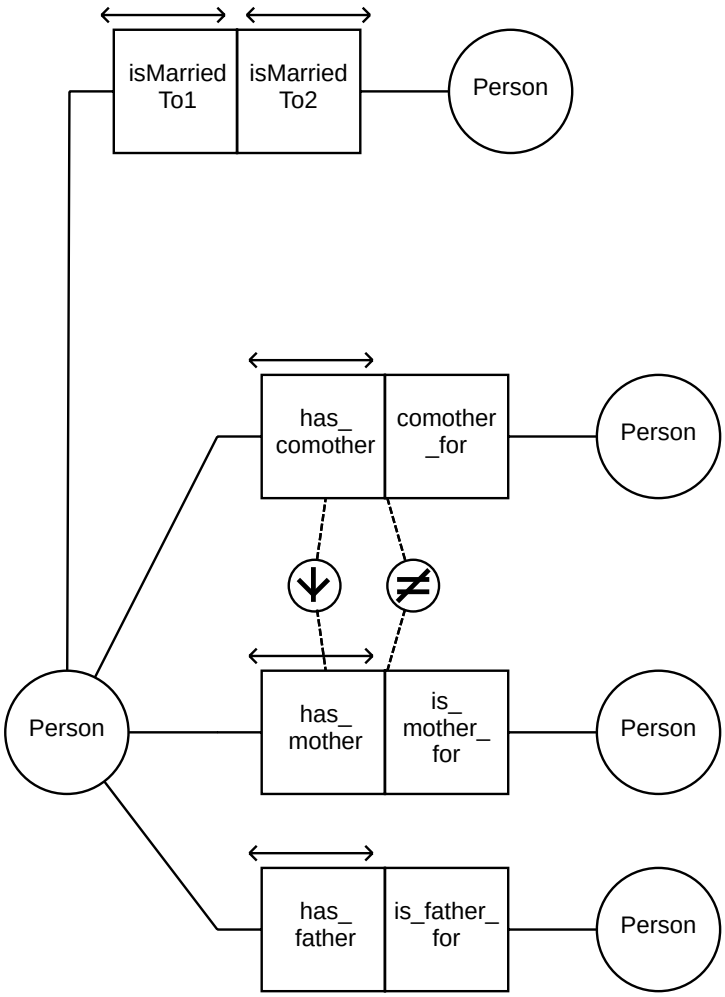
Identification is 11 digits, whether it is a birthnumber or a D-number.
The first 6 digits are either date of birth ddmmyy if it is a birthnumber, or the day of birth + 40, and then mmyy.
The last 5 digits are calculated controll digits. In the case of a birthnumber the birth date on this model is redundant.


FirstNa
me
is
for
Middle
Name
is
for
name
LastNa
me
is
for


Identific
ation
is
for
pid
Year
is
for
YYYY
Month
is
for
MM
DayOf
Month
is
for
DD


has_
place_of_
birth
is_
location
PlaceO
fBirth
has_
marital_
status
is_status
Marital
Status
Person
lives_in
associate
d_with
Address
has_
gender
gender_
for
Gender


PlaceO
fBirth
is
for
location
Marital
Status
is
for
status
Gender
is
for
type



It is not practical from a implementation point of view to have a mandatory role on mother and father.

There is supposed to be an irreflexive ring constraint on the role pairs:

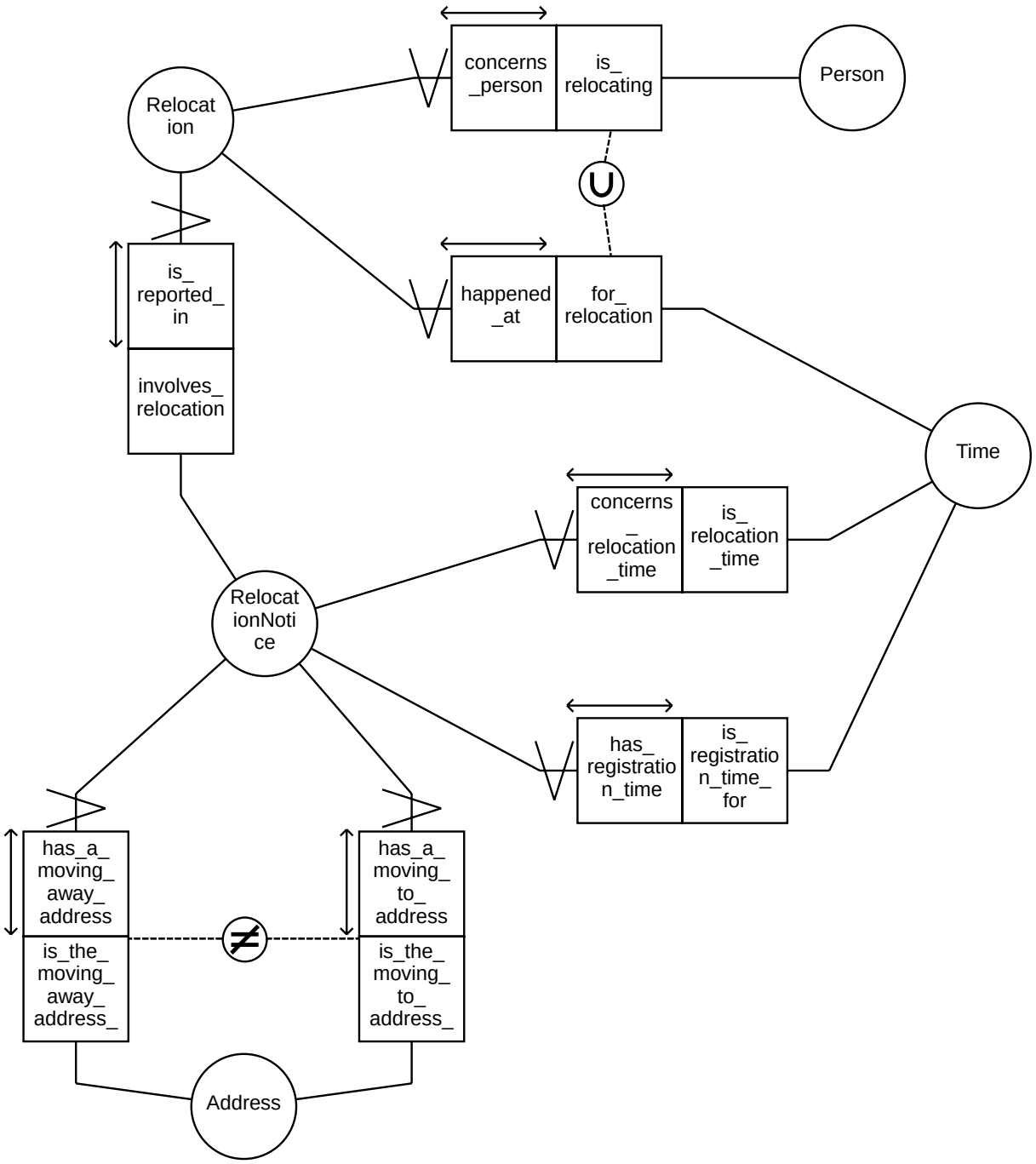
isMarriedTo1 - isMarriedTo2,
has_comother - comother_for,
has_mother - is_mother_for,
has_father - is_father_for.

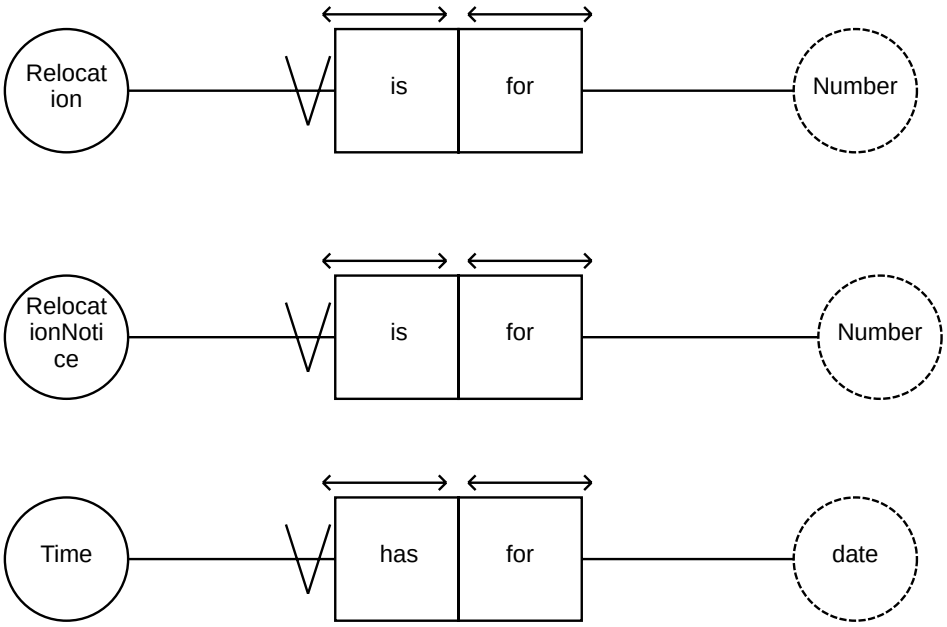
In addition there is supposed to be a symmetric ring constraint

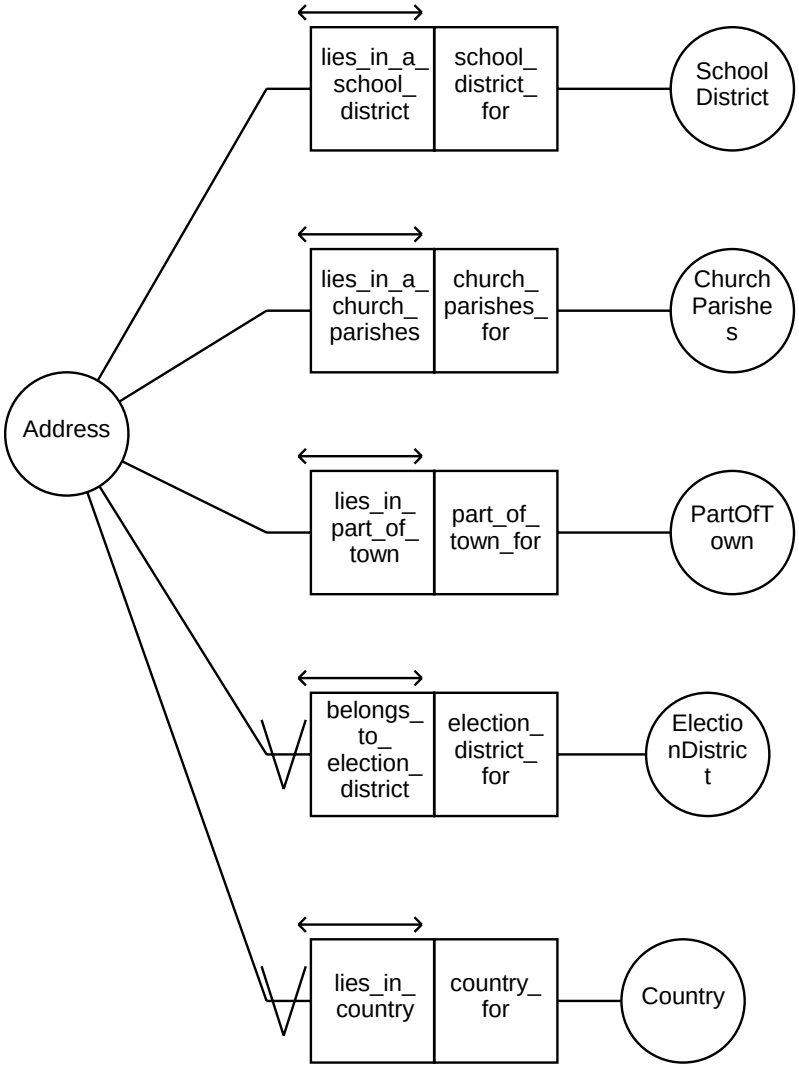
on the role pair:
isMarriedTo1 - isMarriedTo2

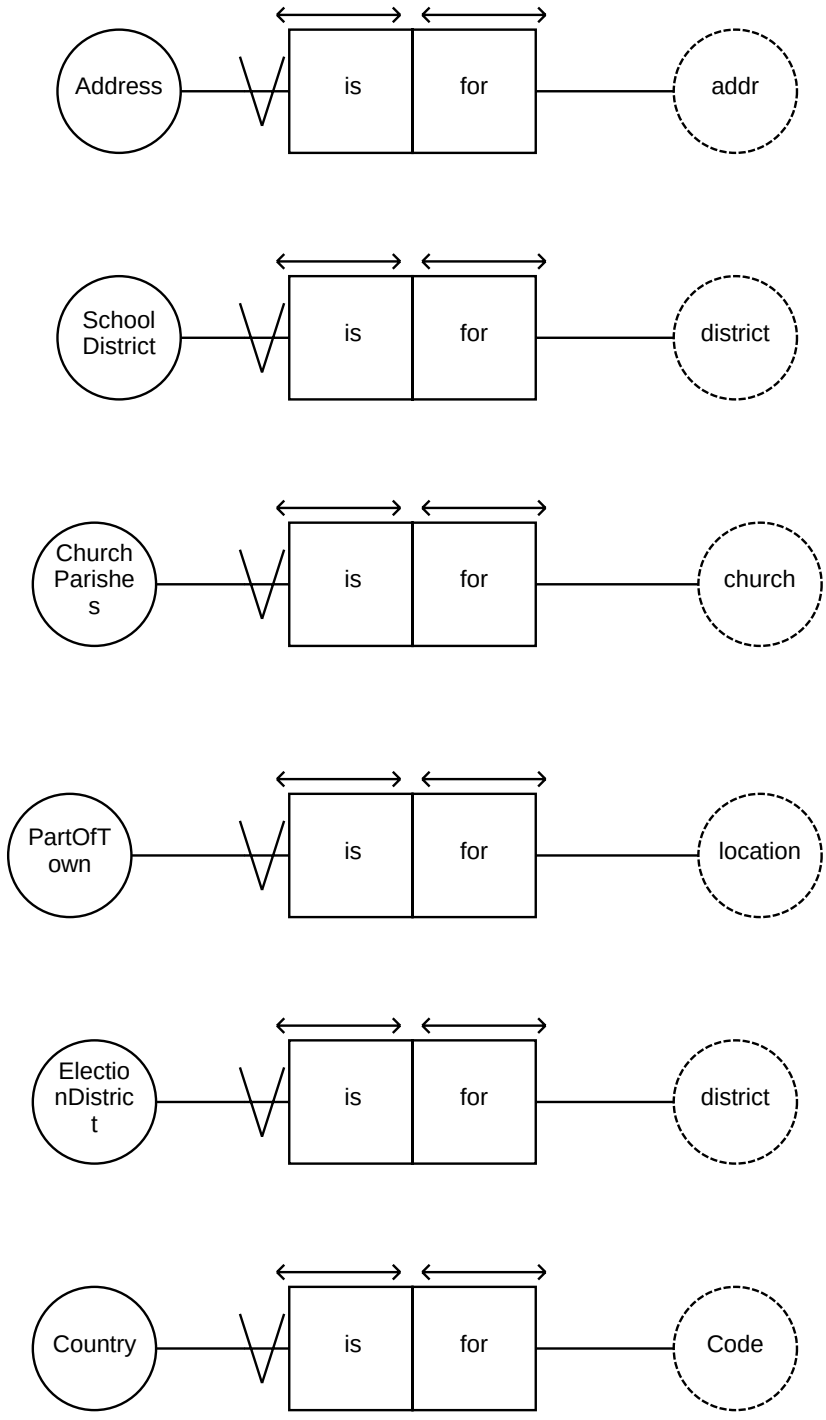
Equivalence of Path.
Relocation - happend_at - Time
and
Relocation - is_reported_in - RelocationNotice - concerns_relocation_time - Time.

Can not be expressed in stORM.




Relocat
ion
is
for
Number
Relocat
ionNoti
ce
is
for
Number
Time
has
for
date


Address
lies_in_a_school_district
school_district_for
School District
lies_in_a_church_parishes
church_parishes_for
Church Parishes
lies_in_part_of_town
part_of_town_for
PartOfTown
belongs_to_election_district
election_district_for
ElectionDistrict
lies_in_country
country_for
Country


Address
is
for
addr
School District
is
for
district
Church Parishes
is
for
church
PartOfTown
is
for
location
ElectionDistrict
is
for
district
Country
is
for
Code

Appendix B

The Database Schema and Tables

B.1 The Database Schema

```
/*************************************************************************
* PostgreSQL Database Description File for databaseSchema
* Generated by stORM © Norsync AS 2008
* Generation Date is 2011/10/27
* Model Number is 13
* Version Checksum is 88146
*************************************************************************/

/*************************************************************************
* Table Definitions
*************************************************************************/

CREATE TABLE Address (
   addr                 VARCHAR(20) NOT NULL,
   country              VARCHAR(20) NOT NULL,
   electionDistrict     VARCHAR(20) NOT NULL,
   churchParish         VARCHAR(20),
   partOfTown           VARCHAR(20),
   schoolDistrict       VARCHAR(20)
);

CREATE TABLE ChurchParishes (
   church               VARCHAR(20) NOT NULL
);

CREATE TABLE Country (
   country              VARCHAR(20) NOT NULL
);

CREATE TABLE DateOfBirth (
   year                 VARCHAR(20) NOT NULL,
   month                VARCHAR(20) NOT NULL,
   day                  VARCHAR(20) NOT NULL
);

CREATE TABLE DayOfMonth (
   day                  VARCHAR(20) NOT NULL
);

CREATE TABLE ElectionDistrict (
   electionDistrict     VARCHAR(20) NOT NULL
);

CREATE TABLE FirstName (
   firstname           VARCHAR(20) NOT NULL
);

CREATE TABLE Gender (
   gender               VARCHAR(20) NOT NULL
);

CREATE TABLE Identification (
   pid                  VARCHAR(20) NOT NULL
);

CREATE TABLE LastName (
   lastname            VARCHAR(20) NOT NULL
);

CREATE TABLE MaritalStatus (
   maritalStatus       VARCHAR(20) NOT NULL
```

```
);

CREATE TABLE MiddleName (
   middlename          VARCHAR(20) NOT NULL
);

CREATE TABLE Month (
   month               VARCHAR(20) NOT NULL
);

CREATE TABLE PartOfTown (
   partOfTown          VARCHAR(20) NOT NULL
);

CREATE TABLE Person (
   pid                 VARCHAR(20) NOT NULL,
   year                VARCHAR(20) NOT NULL,
   month               VARCHAR(20) NOT NULL,
   day                 VARCHAR(20) NOT NULL,
   gender              VARCHAR(20) NOT NULL,
   lastname            VARCHAR(20) NOT NULL,
   maritalStatus       VARCHAR(20) NOT NULL,
   placeOfBirth        VARCHAR(20) NOT NULL,
   addr                VARCHAR(20),
   comother            VARCHAR(20),
   father              VARCHAR(20),
   mother              VARCHAR(20)
);

CREATE TABLE Person_FirstName (
   pid                 VARCHAR(20) NOT NULL,
   firstname           VARCHAR(20) NOT NULL
);

CREATE TABLE Person_MiddleName (
   pid                 VARCHAR(20) NOT NULL,
   middlename          VARCHAR(20) NOT NULL
);

CREATE TABLE Marriage (
   isMarriedTo1        VARCHAR(20) NOT NULL,
   isMarriedTo2        VARCHAR(20) NOT NULL
);

CREATE TABLE PlaceOfBirth (
   placeOfBirth        VARCHAR(20) NOT NULL
);

CREATE TABLE Relocation (
   number              VARCHAR(20) NOT NULL,
   pid                 VARCHAR(20) NOT NULL,
   relocationDate      VARCHAR(20) NOT NULL,
   relocationNotice    VARCHAR(20) NOT NULL
);

CREATE TABLE RelocationNotice (
   number              VARCHAR(20) NOT NULL,
   movingAwayAddr      VARCHAR(20) NOT NULL,
   movingToAddr        VARCHAR(20) NOT NULL,
   registrationDate    VARCHAR(20) NOT NULL,
   relocationDate      VARCHAR(20) NOT NULL
```

```
);

CREATE TABLE SchoolDistrict (
   schoolDistrict       VARCHAR(20) NOT NULL
);

CREATE TABLE Time (
   date                 VARCHAR(20) NOT NULL
);

CREATE TABLE Year (
   year                 VARCHAR(20) NOT NULL
);


/*==============================================================
* SERIAL columns
==============================================================*/


/*==============================================================
* Primary and Unique Key Definitions
==============================================================*/

ALTER TABLE Address ADD PRIMARY KEY (addr);

ALTER TABLE ChurchParishes ADD PRIMARY KEY (church);

ALTER TABLE Country ADD PRIMARY KEY (country);

ALTER TABLE DateOfBirth ADD PRIMARY KEY (year,month,day);

ALTER TABLE DayOfMonth ADD PRIMARY KEY (day);

ALTER TABLE ElectionDistrict ADD PRIMARY KEY (electionDistrict);

ALTER TABLE FirstName ADD PRIMARY KEY (firstname);

ALTER TABLE Gender ADD PRIMARY KEY (gender);

ALTER TABLE Identification ADD PRIMARY KEY (pid);

ALTER TABLE LastName ADD PRIMARY KEY (lastname);

ALTER TABLE MaritalStatus ADD PRIMARY KEY (maritalStatus);

ALTER TABLE MiddleName ADD PRIMARY KEY (middlename);

ALTER TABLE Month ADD PRIMARY KEY (month);

ALTER TABLE PartOfTown ADD PRIMARY KEY (partOfTown);

ALTER TABLE Person ADD PRIMARY KEY (pid);

ALTER TABLE Person_FirstName ADD PRIMARY KEY (pid,firstname);

ALTER TABLE Person_MiddleName ADD PRIMARY KEY (pid,middlename);

ALTER TABLE Marriage ADD PRIMARY KEY (isMarriedTo1);

ALTER TABLE Marriage ADD UNIQUE (isMarriedTo2);
```

```
ALTER TABLE PlaceOfBirth ADD PRIMARY KEY (placeOfBirth);

ALTER TABLE Relocation ADD PRIMARY KEY (number);

ALTER TABLE Relocation ADD UNIQUE (pid,relocationDate);

ALTER TABLE RelocationNotice ADD PRIMARY KEY (number);

ALTER TABLE SchoolDistrict ADD PRIMARY KEY (schoolDistrict);

ALTER TABLE Time ADD PRIMARY KEY (date);

ALTER TABLE Year ADD PRIMARY KEY (year);


/*****************************************************************
* Foreign Key Constraints
*****************************************************************/

ALTER TABLE Address ADD CONSTRAINT Country_Address
   FOREIGN KEY (country) REFERENCES Country (country);

ALTER TABLE Address ADD CONSTRAINT ElectionDistrict_Address
   FOREIGN KEY (electionDistrict) REFERENCES ElectionDistrict (electionDistrict);

ALTER TABLE Address ADD CONSTRAINT ChurchParishes_Address
   FOREIGN KEY (churchParish) REFERENCES ChurchParishes (church);

ALTER TABLE Address ADD CONSTRAINT PartOfTown_Address
   FOREIGN KEY (partOfTown) REFERENCES PartOfTown (partOfTown);

ALTER TABLE Address ADD CONSTRAINT SchoolDistrict_Address
   FOREIGN KEY (schoolDistrict) REFERENCES SchoolDistrict (schoolDistrict);

ALTER TABLE DateOfBirth ADD CONSTRAINT Year_DateOfBirth
   FOREIGN KEY (year) REFERENCES Year (year);

ALTER TABLE DateOfBirth ADD CONSTRAINT Month_DateOfBirth
   FOREIGN KEY (month) REFERENCES Month (month);

ALTER TABLE DateOfBirth ADD CONSTRAINT DayOfMonth_DateOfBirth
   FOREIGN KEY (day) REFERENCES DayOfMonth (day);

ALTER TABLE Person ADD CONSTRAINT Identification_Person
   FOREIGN KEY (pid) REFERENCES Identification (pid);

ALTER TABLE Person ADD CONSTRAINT DateOfBirth_Person
   FOREIGN KEY (year,month,day) REFERENCES DateOfBirth (year,month,day);

ALTER TABLE Person ADD CONSTRAINT Gender_Person
   FOREIGN KEY (gender) REFERENCES Gender (gender);

ALTER TABLE Person ADD CONSTRAINT LastName_Person
   FOREIGN KEY (lastname) REFERENCES LastName (lastname);

ALTER TABLE Person ADD CONSTRAINT MaritalStatus_Person
   FOREIGN KEY (maritalStatus) REFERENCES MaritalStatus (maritalStatus);

ALTER TABLE Person ADD CONSTRAINT PlaceOfBirth_Person
   FOREIGN KEY (placeOfBirth) REFERENCES PlaceOfBirth (placeOfBirth);
```

```
ALTER TABLE Person ADD CONSTRAINT Address_Person
   FOREIGN KEY (addr) REFERENCES Address (addr);

ALTER TABLE Person ADD CONSTRAINT Person_Person
   FOREIGN KEY (comother) REFERENCES Person (pid);

ALTER TABLE Person ADD CONSTRAINT Person_Person1
   FOREIGN KEY (father) REFERENCES Person (pid);

ALTER TABLE Person ADD CONSTRAINT Person_Person2
   FOREIGN KEY (mother) REFERENCES Person (pid);

ALTER TABLE Person_FirstName ADD CONSTRAINT Person_Person_FirstName
   FOREIGN KEY (pid) REFERENCES Person (pid);

ALTER TABLE Person_FirstName ADD CONSTRAINT FirstName_Person_Fire
   FOREIGN KEY (firstname) REFERENCES FirstName (firstname);

ALTER TABLE Person_MiddleName ADD CONSTRAINT Person_Person_MiddleName
   FOREIGN KEY (pid) REFERENCES Person (pid);

ALTER TABLE Person_MiddleName ADD CONSTRAINT MiddleName_Person_Mide
   FOREIGN KEY (middlename) REFERENCES MiddleName (middlename);

ALTER TABLE Marriage ADD CONSTRAINT Person_Person_Person
   FOREIGN KEY (isMarriedTo1) REFERENCES Person (pid);

ALTER TABLE Marriage ADD CONSTRAINT Person_Person_Person1
   FOREIGN KEY (isMarriedTo2) REFERENCES Person (pid);

ALTER TABLE Relocation ADD CONSTRAINT Person_Relocation
   FOREIGN KEY (pid) REFERENCES Person (pid);

ALTER TABLE Relocation ADD CONSTRAINT Time_Relocation
   FOREIGN KEY (relocationDate) REFERENCES Time (date);

ALTER TABLE Relocation ADD CONSTRAINT Relocatione_Relocation
   FOREIGN KEY (relocationNotice) REFERENCES RelocationNotice (number);

ALTER TABLE RelocationNotice ADD CONSTRAINT Address_RelocationNotice
   FOREIGN KEY (movingAwayAddr) REFERENCES Address (addr);

ALTER TABLE RelocationNotice ADD CONSTRAINT Address_RelocationNoti_A
   FOREIGN KEY (movingToAddr) REFERENCES Address (addr);

ALTER TABLE RelocationNotice ADD CONSTRAINT Time_RelocationNotice
   FOREIGN KEY (registrationDate) REFERENCES Time (date);

ALTER TABLE RelocationNotice ADD CONSTRAINT Time_RelocationNotice1
   FOREIGN KEY (relocationDate) REFERENCES Time (date);


/*************************************************************
* Index Definitions
*************************************************************/

CREATE INDEX Country_Address ON               Address (country);

CREATE INDEX ElectionDistrict_Address ON      Address (electionDistrict);
```

```
CREATE INDEX ChurchParishes_Address ON        Address (churchParish);
CREATE INDEX PartOfTown_Address ON            Address (partOfTown);
CREATE INDEX SchoolDistrict_Address ON        Address (schoolDistrict);
CREATE INDEX Year_DateOfBirth ON              DateOfBirth (year);
CREATE INDEX Month_DateOfBirth ON             DateOfBirth (month);
CREATE INDEX DayOfMonth_DateOfBirth ON        DateOfBirth (day);
CREATE INDEX DateOfBirth_Person ON            Person (year,month,day);
CREATE INDEX Gender_Person ON                 Person (gender);
CREATE INDEX LastName_Person ON               Person (lastname);
CREATE INDEX MaritalStatus_Person ON          Person (maritalStatus);
CREATE INDEX PlaceOfBirth_Person ON           Person (placeOfBirth);
CREATE INDEX Address_Person ON                Person (addr);
CREATE INDEX Person_Person ON                 Person (comother);
CREATE INDEX Person_Person1 ON                Person (father);
CREATE INDEX Person_Person2 ON                Person (mother);
CREATE INDEX Exclud_1 ON                      Person (pid,comother);
CREATE INDEX Exclud_1_A ON                    Person (pid,mother);
CREATE INDEX Person_Person_FirstName ON       Person_FirstName (pid);
CREATE INDEX FirstName_Person_Fire ON         Person_FirstName (firstname);
CREATE INDEX Person_Person_MiddleName ON      Person_MiddleName (pid);
CREATE INDEX MiddleName_Person_Mide ON        Person_MiddleName (middlename);
CREATE INDEX Person_Relocation ON             Relocation (pid);
CREATE INDEX Time_Relocation ON               Relocation (relocationDate);
CREATE INDEX Relocatione_Relocation ON        Relocation (relocationNotice);
CREATE INDEX Address_RelocationNotice ON      RelocationNotice (movingAwayAddr);
CREATE INDEX Address_RelocationNoti_A ON      RelocationNotice (movingToAddr);
CREATE INDEX Time_RelocationNotice ON         RelocationNotice (registrationDate);
CREATE INDEX Time_RelocationNotice1 ON        RelocationNotice (relocationDate);
CREATE INDEX Exclud_3 ON                      RelocationNotice (number,movingAwayAddr);
CREATE INDEX Exclud_3_A ON                    RelocationNotice (number,movingToAddr);
```

```
/* UNSUPPORTED CONSTRAINT: Referential Exclude "Exclud_1" */

/* UNSUPPORTED CONSTRAINT: Referential Exclude "Exclud_3" */

/* UNSUPPORTED CONSTRAINT: Intratable Subset "DiaSu_1" */
```

B.2 The Database Tables

Field	Type	Null	Key	Default	Extra
addr	varchar(20)	NO	PRI	NULL	
country	varchar(20)	NO	MUL	NULL	
electionDistrict	varchar(20)	NO	MUL	NULL	
churchParish	varchar(20)	YES	MUL	NULL	
partOfTown	varchar(20)	YES	MUL	NULL	
schoolDistrict	varchar(20)	YES	MUL	NULL	

Figure B.1: Address database table.

Field	Type	Null	Key	Default	Extra
church	varchar(20)	NO	PRI	NULL	

Figure B.2: ChurchParishes database table.

Field	Type	Null	Key	Default	Extra
country	varchar(20)	NO	PRI	NULL	

Figure B.3: Country database table.

Field	Type	Null	Key	Default	Extra
year	varchar(20)	NO	PRI	NULL	
month	varchar(20)	NO	PRI	NULL	
day	varchar(20)	NO	PRI	NULL	

Figure B.4: DateOfBirth database table.

Field	Type	Null	Key	Default	Extra
day	varchar(20)	NO	PRI	NULL	

Figure B.5: DayOfMonth database table.

```
+------------------+-------------+------+-----+---------+-------+
| Field            | Type        | Null | Key | Default | Extra |
+------------------+-------------+------+-----+---------+-------+
| electionDistrict | varchar(20) | NO   | PRI | NULL    |       |
+------------------+-------------+------+-----+---------+-------+
```

Figure B.6: ElectionDistrict database table.

```
+-----------+-------------+------+-----+---------+-------+
| Field     | Type        | Null | Key | Default | Extra |
+-----------+-------------+------+-----+---------+-------+
| firstname | varchar(20) | NO   | PRI | NULL    |       |
+-----------+-------------+------+-----+---------+-------+
```

Figure B.7: FirstName database table.

```
+--------+-------------+------+-----+---------+-------+
| Field  | Type        | Null | Key | Default | Extra |
+--------+-------------+------+-----+---------+-------+
| gender | varchar(20) | NO   | PRI | NULL    |       |
+--------+-------------+------+-----+---------+-------+
```

Figure B.8: Gender database table.

```
+-------+-------------+------+-----+---------+-------+
| Field | Type        | Null | Key | Default | Extra |
+-------+-------------+------+-----+---------+-------+
| pid   | varchar(20) | NO   | PRI | NULL    |       |
+-------+-------------+------+-----+---------+-------+
```

Figure B.9: Identification database table.

```
+----------+-------------+------+-----+---------+-------+
| Field    | Type        | Null | Key | Default | Extra |
+----------+-------------+------+-----+---------+-------+
| lastname | varchar(20) | NO   | PRI | NULL    |       |
+----------+-------------+------+-----+---------+-------+
```

Figure B.10: LastName database table.

```
+---------------+-------------+------+-----+---------+-------+
| Field         | Type        | Null | Key | Default | Extra |
+---------------+-------------+------+-----+---------+-------+
| maritalStatus | varchar(20) | NO   | PRI | NULL    |       |
+---------------+-------------+------+-----+---------+-------+
```

Figure B.11: MaritalStatus database table.

Field	Type	Null	Key	Default	Extra
isMarriedTo1	varchar(20)	NO	PRI	NULL	
isMarriedTo2	varchar(20)	NO	UNI	NULL	

Figure B.12: Marriage database table.

Field	Type	Null	Key	Default	Extra
middlename	varchar(20)	NO	PRI	NULL	

Figure B.13: MiddleName database table.

Field	Type	Null	Key	Default	Extra
month	varchar(20)	NO	PRI	NULL	

Figure B.14: Month database table.

Field	Type	Null	Key	Default	Extra
partOfTown	varchar(20)	NO	PRI	NULL	

Figure B.15: PartOfTown database table.

Field	Type	Null	Key	Default	Extra
pid	varchar(20)	NO	PRI	NULL	
year	varchar(20)	NO	MUL	NULL	
month	varchar(20)	NO		NULL	
day	varchar(20)	NO		NULL	
gender	varchar(20)	NO	MUL	NULL	
lastname	varchar(20)	NO	MUL	NULL	
maritalStatus	varchar(20)	NO	MUL	NULL	
placeOfBirth	varchar(20)	NO	MUL	NULL	
addr	varchar(20)	YES	MUL	NULL	
comother	varchar(20)	YES	MUL	NULL	
father	varchar(20)	YES	MUL	NULL	
mother	varchar(20)	YES	MUL	NULL	

Figure B.16: Person database table.

```
+-----------+-------------+------+-----+---------+-------+
| Field     | Type        | Null | Key | Default | Extra |
+-----------+-------------+------+-----+---------+-------+
| pid       | varchar(20) | NO   | PRI | NULL    |       |
| firstname | varchar(20) | NO   | PRI | NULL    |       |
+-----------+-------------+------+-----+---------+-------+
```

Figure B.17: Person_FirstName database table.

```
+------------+-------------+------+-----+---------+-------+
| Field      | Type        | Null | Key | Default | Extra |
+------------+-------------+------+-----+---------+-------+
| pid        | varchar(20) | NO   | PRI | NULL    |       |
| middlename | varchar(20) | NO   | PRI | NULL    |       |
+------------+-------------+------+-----+---------+-------+
```

Figure B.18: Person_MiddleName database table.

```
+--------------+-------------+------+-----+---------+-------+
| Field        | Type        | Null | Key | Default | Extra |
+--------------+-------------+------+-----+---------+-------+
| placeOfBirth | varchar(20) | NO   | PRI | NULL    |       |
+--------------+-------------+------+-----+---------+-------+
```

Figure B.19: PlaceOfBirth database table.

```
+------------------+-------------+------+-----+---------+-------+
| Field            | Type        | Null | Key | Default | Extra |
+------------------+-------------+------+-----+---------+-------+
| number           | varchar(20) | NO   | PRI | NULL    |       |
| pid              | varchar(20) | NO   | MUL | NULL    |       |
| relocationDate   | varchar(20) | NO   | MUL | NULL    |       |
| relocationNotice | varchar(20) | NO   | MUL | NULL    |       |
+------------------+-------------+------+-----+---------+-------+
```

Figure B.20: Relocation database table.

```
+------------------+-------------+------+-----+---------+-------+
| Field            | Type        | Null | Key | Default | Extra |
+------------------+-------------+------+-----+---------+-------+
| number           | varchar(20) | NO   | PRI | NULL    |       |
| movingAwayAddr   | varchar(20) | NO   | MUL | NULL    |       |
| movingToAddr     | varchar(20) | NO   | MUL | NULL    |       |
| registrationDate | varchar(20) | NO   | MUL | NULL    |       |
| relocationDate   | varchar(20) | NO   | MUL | NULL    |       |
+------------------+-------------+------+-----+---------+-------+
```

Figure B.21: RelocationNotice database table.

Field	Type	Null	Key	Default	Extra
schoolDistrict	varchar(20)	NO	PRI	NULL	

Figure B.22: SchoolDistrict database table.

Field	Type	Null	Key	Default	Extra
date	varchar(20)	NO	PRI	NULL	

Figure B.23: Time database table.

Field	Type	Null	Key	Default	Extra
year	varchar(20)	NO	PRI	NULL	

Figure B.24: Year database table.

Appendix C

The Complete OWL Model of the Norwegian National Register

```
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix : <http://sws.ifi.uio.no/vocab/dsf#> .
@prefix xml: <http://www.w3.org/XML/1998/namespace> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix skos: <http://www.w3.org/2004/02/skos/core#> .
@base <http://sws.ifi.uio.no/vocab/dsf> .

<http://sws.ifi.uio.no/vocab/dsf> rdf:type owl:Ontology ;

                                  rdfs:comment "work by Charlotte Løvdahl" .


#################################################################
#
#    Annotation properties
#
#################################################################


rdfs:comment rdf:type owl:AnnotationProperty .



#################################################################
#
#    Object Properties
#
#################################################################


###  http://sws.ifi.uio.no/vocab/dsf#addressAssociatedWithPerson

:addressAssociatedWithPerson rdf:type owl:ObjectProperty ;

                             rdfs:domain :Address ;

                             rdfs:range :Person ;

                             owl:inverseOf :personLivesInAddress .


###  http://sws.ifi.uio.no/vocab/dsf#addressBelongsToElectionDistrict

:addressBelongsToElectionDistrict rdf:type owl:ObjectProperty ;

                                  rdfs:domain :Address ;

                                  rdfs:range :ElectionDistrict .


###  http://sws.ifi.uio.no/vocab/dsf#addressLiesInChurchParishes

:addressLiesInChurchParishes rdf:type owl:ObjectProperty ;

                             rdfs:domain :Address ;

                             rdfs:range :ChurchParishes .
```

```
###  http://sws.ifi.uio.no/vocab/dsf#addressLiesInCountry

:addressLiesInCountry rdf:type owl:ObjectProperty ;
                      rdfs:domain :Address ;
                      rdfs:range :Country .

###  http://sws.ifi.uio.no/vocab/dsf#addressLiesInPartOfTown

:addressLiesInPartOfTown rdf:type owl:ObjectProperty ;
                         rdfs:domain :Address ;
                         rdfs:range :PartOfTown ;
                         owl:inverseOf :partOfTownForAddress .

###  http://sws.ifi.uio.no/vocab/dsf#addressLiesInSchoolDistrict

:addressLiesInSchoolDistrict rdf:type owl:ObjectProperty .

###  http://sws.ifi.uio.no/vocab/dsf#churchParishesForAddress

:churchParishesForAddress rdf:type owl:ObjectProperty ;
                          owl:inverseOf :addressLiesInChurchParishes .

###  http://sws.ifi.uio.no/vocab/dsf#countryForAddress

:countryForAddress rdf:type owl:ObjectProperty ;
                   owl:inverseOf :addressLiesInCountry .

###  http://sws.ifi.uio.no/vocab/dsf#dateOfBirthContainsDay

:dateOfBirthContainsDay rdf:type owl:ObjectProperty ;
                        rdfs:domain :DateOfBirth ;
                        rdfs:range :DayOfMonth ;
                        owl:inverseOf :dayIsPartOfDateOfBirth .

###  http://sws.ifi.uio.no/vocab/dsf#dateOfBirthContainsMonth
```

```
:dateOfBirthContainsMonth rdf:type owl:ObjectProperty ;
                          rdfs:domain :DateOfBirth ;
                          rdfs:range :Month ;
                          owl:inverseOf :monthIsPartOfDateOfBirth .

###  http://sws.ifi.uio.no/vocab/dsf#dateofBirthContainsYear
:dateofBirthContainsYear rdf:type owl:ObjectProperty .

###  http://sws.ifi.uio.no/vocab/dsf#dayIsPartOfDateOfBirth
:dayIsPartOfDateOfBirth rdf:type owl:ObjectProperty .

###  http://sws.ifi.uio.no/vocab/dsf#electionDistrictForAddress
:electionDistrictForAddress rdf:type owl:ObjectProperty ;
                            owl:inverseOf :addressBelongsToElectionDistrict .

###  http://sws.ifi.uio.no/vocab/dsf#isDateOfBirthForPerson
:isDateOfBirthForPerson rdf:type owl:ObjectProperty ;
                        rdfs:domain :DateOfBirth ;
                        rdfs:range :Person ;
                        owl:inverseOf :personHasDateOfBirth .

###  http://sws.ifi.uio.no/vocab/dsf#isFirstNameForPerson
:isFirstNameForPerson rdf:type owl:ObjectProperty ;
                      rdfs:domain :FirstName ;
                      rdfs:range :Person ;
                      owl:inverseOf :personHasFirstName .

###  http://sws.ifi.uio.no/vocab/dsf#isGenderForPerson
:isGenderForPerson rdf:type owl:ObjectProperty ;
                   rdfs:domain :Gender ;
                   rdfs:range :Person ;
```

```
                    owl:inverseOf :personHasGender .


###  http://sws.ifi.uio.no/vocab/dsf#isIdForPerson
:isIdForPerson rdf:type owl:ObjectProperty ;
               rdfs:domain :Identification ;
               rdfs:range :Person ;
               owl:inverseOf :personHasId .


###  http://sws.ifi.uio.no/vocab/dsf#isLastNameForPerson
:isLastNameForPerson rdf:type owl:ObjectProperty ;
                     rdfs:domain :LastName ;
                     rdfs:range :Person ;
                     owl:inverseOf :personHasLastName .


###  http://sws.ifi.uio.no/vocab/dsf#isMaritalStatusForPerson
:isMaritalStatusForPerson rdf:type owl:ObjectProperty ;
                          owl:inverseOf :personHasMaritalStatus .


###  http://sws.ifi.uio.no/vocab/dsf#isMiddleNameForPerson
:isMiddleNameForPerson rdf:type owl:ObjectProperty .


###  http://sws.ifi.uio.no/vocab/dsf#isMovingAwayAddressForRelocationNotice
:isMovingAwayAddressForRelocationNotice rdf:type owl:ObjectProperty ;

owl:propertyDisjointWith :isMovingToAddressForRelocationNotice ;

owl:inverseOf :relocationNoticeHasMovingAwayAddress .


###  http://sws.ifi.uio.no/vocab/dsf#isMovingToAddressForRelocationNotice
:isMovingToAddressForRelocationNotice rdf:type owl:ObjectProperty ;

owl:inverseOf :relocationNoticeHasMovingToAddress .
```

```
###  http://sws.ifi.uio.no/vocab/dsf#isPlaceOfBirthForPerson

:isPlaceOfBirthForPerson rdf:type owl:ObjectProperty ;
                         owl:inverseOf :personHasPlaceOfBirth .


###  http://sws.ifi.uio.no/vocab/dsf#monthIsPartOfDateOfBirth

:monthIsPartOfDateOfBirth rdf:type owl:ObjectProperty .


###  http://sws.ifi.uio.no/vocab/dsf#partOfTownForAddress

:partOfTownForAddress rdf:type owl:ObjectProperty ;
                      rdfs:range :Address ;
                      rdfs:domain :PartOfTown .


###  http://sws.ifi.uio.no/vocab/dsf#personHasCoMother

:personHasCoMother rdf:type owl:IrreflexiveProperty ,
                            owl:ObjectProperty .


###  http://sws.ifi.uio.no/vocab/dsf#personHasDateOfBirth

:personHasDateOfBirth rdf:type owl:ObjectProperty .


###  http://sws.ifi.uio.no/vocab/dsf#personHasFather

:personHasFather rdf:type owl:IrreflexiveProperty ,
                          owl:ObjectProperty .


###  http://sws.ifi.uio.no/vocab/dsf#personHasFirstName

:personHasFirstName rdf:type owl:ObjectProperty .


###  http://sws.ifi.uio.no/vocab/dsf#personHasGender

:personHasGender rdf:type owl:ObjectProperty .


###  http://sws.ifi.uio.no/vocab/dsf#personHasId
```

```
:personHasId rdf:type owl:ObjectProperty .


###  http://sws.ifi.uio.no/vocab/dsf#personHasLastName

:personHasLastName rdf:type owl:ObjectProperty .


###  http://sws.ifi.uio.no/vocab/dsf#personHasMaritalStatus

:personHasMaritalStatus rdf:type owl:ObjectProperty ;

                        rdfs:range :MaritalStatus ;

                        rdfs:domain :Person .


###  http://sws.ifi.uio.no/vocab/dsf#personHasMiddleName

:personHasMiddleName rdf:type owl:ObjectProperty ;

                     rdfs:range :MiddleName ;

                     rdfs:domain :Person ;

                     owl:inverseOf :isMiddleNameForPerson .


###  http://sws.ifi.uio.no/vocab/dsf#personHasMother

:personHasMother rdf:type owl:IrreflexiveProperty ,
                          owl:ObjectProperty ;

                 rdfs:domain :Person ;

                 rdfs:range :Person .


###  http://sws.ifi.uio.no/vocab/dsf#personHasPlaceOfBirth

:personHasPlaceOfBirth rdf:type owl:ObjectProperty ;

                       rdfs:domain :Person ;

                       rdfs:range :PlaceOfBirth .


###  http://sws.ifi.uio.no/vocab/dsf#personIsCoMotherFor

:personIsCoMotherFor rdf:type owl:InverseFunctionalProperty ,
                              owl:IrreflexiveProperty ,
                              owl:ObjectProperty ;

                     rdfs:domain :Person ;
```

```
                     rdfs:range :PersonWithCoMother ;

                     owl:inverseOf :personHasCoMother ;

                     owl:propertyDisjointWith :personIsMotherFor .


###  http://sws.ifi.uio.no/vocab/dsf#personIsFatherFor

:personIsFatherFor rdf:type owl:InverseFunctionalProperty ,
                            owl:IrreflexiveProperty ,
                            owl:ObjectProperty ;

                   rdfs:range :Person ;

                   rdfs:domain :Person ;

                   owl:inverseOf :personHasFather .


###  http://sws.ifi.uio.no/vocab/dsf#personIsMarriedTo

:personIsMarriedTo rdf:type owl:IrreflexiveProperty ,
                            owl:ObjectProperty ,
                            owl:SymmetricProperty ;

                   rdfs:domain :Person ;

                   rdfs:range :Person .


###  http://sws.ifi.uio.no/vocab/dsf#personIsMotherFor

:personIsMotherFor rdf:type owl:InverseFunctionalProperty ,
                            owl:IrreflexiveProperty ,
                            owl:ObjectProperty ;

                   owl:inverseOf :personHasMother .


###  http://sws.ifi.uio.no/vocab/dsf#personIsRelocating

:personIsRelocating rdf:type owl:ObjectProperty ;

                    rdfs:domain :Person ;

                    rdfs:range :Relocation ;

                    owl:inverseOf :relocationConcernsPerson .


###  http://sws.ifi.uio.no/vocab/dsf#personLivesInAddress

:personLivesInAddress rdf:type owl:ObjectProperty .
```

```
###  http://sws.ifi.uio.no/vocab/dsf#relocationConcernsPerson

:relocationConcernsPerson rdf:type owl:ObjectProperty .


###  http://sws.ifi.uio.no/vocab/dsf#relocationHappendAtTime

:relocationHappendAtTime rdf:type owl:ObjectProperty ;
                         rdfs:domain :Relocation ;
                         rdfs:range :Time .


###  http://sws.ifi.uio.no/vocab/dsf#relocationIsReportedInRelocationNotice

:relocationIsReportedInRelocationNotice rdf:type owl:ObjectProperty ;
                                        rdfs:domain :Relocation ;
                                        rdfs:range :RelocationNotice ;

owl:inverseOf :relocationNoticeInvolvesRelocation .


###  http://sws.ifi.uio.no/vocab/dsf#relocationNoticeConcernsRelocationTime

:relocationNoticeConcernsRelocationTime rdf:type owl:ObjectProperty ;
                                        rdfs:domain :RelocationNotice ;
                                        rdfs:range :Time .


###  http://sws.ifi.uio.no/vocab/dsf#relocationNoticeHasMovingAwayAddress

:relocationNoticeHasMovingAwayAddress rdf:type owl:ObjectProperty ;
                                      rdfs:range :Address ;
                                      rdfs:domain :RelocationNotice .


###  http://sws.ifi.uio.no/vocab/dsf#relocationNoticeHasMovingToAddress

:relocationNoticeHasMovingToAddress rdf:type owl:ObjectProperty ;
                                    rdfs:range :Address ;
                                    rdfs:domain :RelocationNotice .


###  http://sws.ifi.uio.no/vocab/dsf#relocationNoticeHasRegistrationTime
```

```
:relocationNoticeHasRegistrationTime rdf:type owl:ObjectProperty ;
                                     rdfs:domain :RelocationNotice ;
                                     rdfs:range :Time .

###  http://sws.ifi.uio.no/vocab/dsf#relocationNoticeInvolvesRelocation
:relocationNoticeInvolvesRelocation rdf:type owl:ObjectProperty .

###  http://sws.ifi.uio.no/vocab/dsf#schoolDistrictForAddress
:schoolDistrictForAddress rdf:type owl:ObjectProperty ;
                          rdfs:range :Address ;
                          rdfs:domain :SchoolDistrict ;
                          owl:inverseOf :addressLiesInSchoolDistrict .

###  http://sws.ifi.uio.no/vocab/dsf#timeIsRegistrationTimeForRelocationNotice
:timeIsRegistrationTimeForRelocationNotice rdf:type owl:ObjectProperty ;
owl:inverseOf :relocationNoticeHasRegistrationTime .

###  http://sws.ifi.uio.no/vocab/dsf#timeIsReloactionTimeForRelocation
:timeIsReloactionTimeForRelocation rdf:type owl:ObjectProperty ;
                                   owl:inverseOf :relocationHappendAtTime .

###  http://sws.ifi.uio.no/vocab/dsf#timeIsRelocationTimeForRelocationNotice
:timeIsRelocationTimeForRelocationNotice rdf:type owl:ObjectProperty ;
owl:inverseOf :relocationNoticeConcernsRelocationTime .

###  http://sws.ifi.uio.no/vocab/dsf#yearIsPartOfDateOfBirth
:yearIsPartOfDateOfBirth rdf:type owl:ObjectProperty ;
                         rdfs:range :DateOfBirth ;
                         rdfs:domain :Year ;
```

```
                        owl:inverseOf :dateofBirthContainsYear .




#################################################################
#
#    Classes
#
#################################################################


###  http://sws.ifi.uio.no/vocab/dsf#Address

:Address rdf:type owl:Class ;

         rdfs:subClassOf [ rdf:type owl:Restriction ;
                           owl:onProperty :addressLiesInPartOfTown ;
                           owl:onClass :PartOfTown ;
                           owl:maxQualifiedCardinality "1"^^xsd:nonNegativeInteger
                         ] ,
                         [ rdf:type owl:Restriction ;
                           owl:onProperty :addressAssociatedWithPerson ;
                           owl:allValuesFrom :Person
                         ] ,
                         [ rdf:type owl:Restriction ;
                           owl:onProperty :isMovingToAddressForRelocationNotice ;
                           owl:allValuesFrom :RelocationNotice
                         ] ,
                         [ rdf:type owl:Restriction ;
                           owl:onProperty :isMovingAwayAddressForRelocationNotice ;
                           owl:allValuesFrom :RelocationNotice
                         ] ,
                         [ rdf:type owl:Restriction ;
                           owl:onProperty :addressLiesInSchoolDistrict ;
                           owl:onClass :SchoolDistrict ;
                           owl:maxQualifiedCardinality "1"^^xsd:nonNegativeInteger
                         ] ,
                         [ rdf:type owl:Restriction ;
                           owl:onProperty :addressLiesInCountry ;
                           owl:onClass :Country ;
                           owl:qualifiedCardinality "1"^^xsd:nonNegativeInteger
                         ] ,
                         [ rdf:type owl:Restriction ;
                           owl:onProperty :addressLiesInChurchParishes ;
                           owl:onClass :ChurchParishes ;
                           owl:maxQualifiedCardinality "1"^^xsd:nonNegativeInteger
                         ] ,
                         [ rdf:type owl:Restriction ;
                           owl:onProperty :addressBelongsToElectionDistrict ;
                           owl:onClass :ElectionDistrict ;
                           owl:qualifiedCardinality "1"^^xsd:nonNegativeInteger
                         ] ;

         owl:hasKey ( :hasAddr
                    ) .
```

```
###  http://sws.ifi.uio.no/vocab/dsf#ChurchParishes

:ChurchParishes rdf:type owl:Class ;

                owl:hasKey ( :hasChurchParish
                           ) .


###  http://sws.ifi.uio.no/vocab/dsf#Country

:Country rdf:type owl:Class ;

         owl:hasKey ( :hasCountry
                    ) .


###  http://sws.ifi.uio.no/vocab/dsf#DateOfBirth

:DateOfBirth rdf:type owl:Class ;

             owl:hasKey ( :dayIsPartOfDateOfBirth
                          :monthIsPartOfDateOfBirth
                          :yearIsPartOfDateOfBirth
                        ) .


###  http://sws.ifi.uio.no/vocab/dsf#DayOfMonth

:DayOfMonth rdf:type owl:Class ;

            owl:hasKey ( :hasDay
                       ) .


###  http://sws.ifi.uio.no/vocab/dsf#ElectionDistrict

:ElectionDistrict rdf:type owl:Class ;

                  owl:hasKey ( :hasElectionDistrict
                             ) .


###  http://sws.ifi.uio.no/vocab/dsf#FirstName

:FirstName rdf:type owl:Class ;

           owl:hasKey ( :hasFirstname
                      ) .


###  http://sws.ifi.uio.no/vocab/dsf#Gender

:Gender rdf:type owl:Class ;

        owl:hasKey ( :hasGender
```

```
                  ) .


###  http://sws.ifi.uio.no/vocab/dsf#Identification

:Identification rdf:type owl:Class .


###  http://sws.ifi.uio.no/vocab/dsf#LastName

:LastName rdf:type owl:Class ;

          owl:hasKey ( :hasLastname
                     ) .


###  http://sws.ifi.uio.no/vocab/dsf#MaritalStatus

:MaritalStatus rdf:type owl:Class ;

               owl:hasKey ( :hasMaritalStatus
                          ) .


###  http://sws.ifi.uio.no/vocab/dsf#MiddleName

:MiddleName rdf:type owl:Class ;

            owl:hasKey ( :hasMiddlename
                       ) .


###  http://sws.ifi.uio.no/vocab/dsf#Month

:Month rdf:type owl:Class ;

       owl:hasKey ( :hasMonth
                  ) .


###  http://sws.ifi.uio.no/vocab/dsf#PartOfTown

:PartOfTown rdf:type owl:Class ;

            owl:hasKey ( :hasPartOfTown
                       ) .


###  http://sws.ifi.uio.no/vocab/dsf#Person

:Person rdf:type owl:Class ;

        rdfs:subClassOf [ rdf:type owl:Restriction ;
                          owl:onProperty :personHasId ;
```

```
  owl:onClass :Identification ;
  owl:qualifiedCardinality "1"^^xsd:nonNegativeInteger
] ,
[ rdf:type owl:Restriction ;
  owl:onProperty :personHasGender ;
  owl:onClass :Gender ;
  owl:qualifiedCardinality "1"^^xsd:nonNegativeInteger
] ,
[ rdf:type owl:Restriction ;
  owl:onProperty :personIsMotherFor ;
  owl:allValuesFrom :PersonWithMother
] ,
[ rdf:type owl:Restriction ;
  owl:onProperty :personIsRelocating ;
  owl:allValuesFrom :Relocation
] ,
[ rdf:type owl:Restriction ;
  owl:onProperty :personHasPlaceOfBirth ;
  owl:onClass :PlaceOfBirth ;
  owl:qualifiedCardinality "1"^^xsd:nonNegativeInteger
] ,
[ rdf:type owl:Restriction ;
  owl:onProperty :personIsCoMotherFor ;
  owl:allValuesFrom :PersonWithCoMother
] ,
[ rdf:type owl:Restriction ;
  owl:onProperty :personHasFirstName ;
  owl:someValuesFrom :FirstName
] ,
[ rdf:type owl:Restriction ;
  owl:onProperty :personHasMiddleName ;
  owl:allValuesFrom :MiddleName
] ,
[ rdf:type owl:Restriction ;
  owl:onProperty :personHasLastName ;
  owl:onClass :LastName ;
  owl:qualifiedCardinality "1"^^xsd:nonNegativeInteger
] ,
[ rdf:type owl:Restriction ;
  owl:onProperty :personHasMaritalStatus ;
  owl:onClass :MaritalStatus ;
  owl:qualifiedCardinality "1"^^xsd:nonNegativeInteger
] ,
[ rdf:type owl:Restriction ;
  owl:onProperty :personHasCoMother ;
  owl:onClass :Person ;
  owl:maxQualifiedCardinality "1"^^xsd:nonNegativeInteger
] ,
[ rdf:type owl:Restriction ;
  owl:onProperty :personHasFather ;
  owl:onClass :Person ;
  owl:maxQualifiedCardinality "1"^^xsd:nonNegativeInteger
] ,
[ rdf:type owl:Restriction ;
  owl:onProperty :personHasMother ;
  owl:onClass :Person ;
  owl:maxQualifiedCardinality "1"^^xsd:nonNegativeInteger
] ,
[ rdf:type owl:Restriction ;
  owl:onProperty :personLivesInAddress ;
  owl:onClass :Address ;
```

```
                        owl:maxQualifiedCardinality "1"^^xsd:nonNegativeInteger
                      ] ,
                      [ rdf:type owl:Restriction ;
                        owl:onProperty :personHasDateOfBirth ;
                        owl:onClass :DateOfBirth ;
                        owl:qualifiedCardinality "1"^^xsd:nonNegativeInteger
                      ] ,
                      [ rdf:type owl:Restriction ;
                        owl:onProperty :personIsFatherFor ;
                        owl:allValuesFrom :Person
                      ] ;

        owl:hasKey ( :personHasId
                   ) .


###  http://sws.ifi.uio.no/vocab/dsf#PersonWithCoMother

:PersonWithCoMother rdf:type owl:Class ;

                    owl:equivalentClass [ rdf:type owl:Restriction ;
                                          owl:onProperty :personHasCoMother ;
                                          owl:someValuesFrom :Person
                                        ] ;

                    rdfs:subClassOf :PersonWithMother .


###  http://sws.ifi.uio.no/vocab/dsf#PersonWithMother

:PersonWithMother rdf:type owl:Class ;

                  owl:equivalentClass [ rdf:type owl:Restriction ;
                                        owl:onProperty :personHasMother ;
                                        owl:someValuesFrom :Person
                                      ] ;

                  rdfs:subClassOf :Person .


###  http://sws.ifi.uio.no/vocab/dsf#PlaceOfBirth

:PlaceOfBirth rdf:type owl:Class ;

              owl:hasKey ( :hasPlaceOfBirth
                         ) .


###  http://sws.ifi.uio.no/vocab/dsf#Relocation

:Relocation rdf:type owl:Class ;

            rdfs:subClassOf [ rdf:type owl:Restriction ;
                              owl:onProperty :relocationConcernsPerson ;
                              owl:onClass :Person ;
                              owl:minQualifiedCardinality "1"^^xsd:nonNegativeInteger
                            ] ,
```

```
                              [ rdf:type owl:Restriction ;
                                owl:onProperty :relocationIsReportedInRelocationNotice ;
                                owl:onClass :RelocationNotice ;
                                owl:qualifiedCardinality "1"^^xsd:nonNegativeInteger
                              ] ,
                              [ rdf:type owl:Restriction ;
                                owl:onProperty :relocationHappendAtTime ;
                                owl:onClass :Time ;
                                owl:qualifiedCardinality "1"^^xsd:nonNegativeInteger
                              ] ;

              owl:hasKey ( :relocationConcernsPerson
                           :relocationHappendAtTime
                         ) ,
                         ( :hasRelocationNr
                         ) .


###  http://sws.ifi.uio.no/vocab/dsf#RelocationNotice

:RelocationNotice rdf:type owl:Class ;

                  rdfs:subClassOf [ rdf:type owl:Restriction ;

owl:onProperty :relocationNoticeConcernsRelocationTime ;
                                    owl:onClass :Time ;
                                    owl:qualifiedCardinality "1"^^xsd:nonNegativeInteger
                                  ] ,
                                  [ rdf:type owl:Restriction ;

owl:onProperty :relocationNoticeHasMovingAwayAddress ;
                                    owl:onClass :Address ;
                                    owl:qualifiedCardinality "1"^^xsd:nonNegativeInteger
                                  ] ,
                                  [ rdf:type owl:Restriction ;
                                    owl:onProperty :relocationNoticeInvolvesRelocation ;
                                    owl:allValuesFrom :Relocation
                                  ] ,
                                  [ rdf:type owl:Restriction ;

owl:onProperty :relocationNoticeHasRegistrationTime ;
                                    owl:onClass :Time ;
                                    owl:qualifiedCardinality "1"^^xsd:nonNegativeInteger
                                  ] ,
                                  [ rdf:type owl:Restriction ;
                                    owl:onProperty :relocationNoticeHasMovingToAddress ;
                                    owl:onClass :Address ;
                                    owl:qualifiedCardinality "1"^^xsd:nonNegativeInteger
                                  ] ;

                  owl:hasKey ( :hasRelocationNoticeNr
                             ) .


###  http://sws.ifi.uio.no/vocab/dsf#SchoolDistrict

:SchoolDistrict rdf:type owl:Class ;

                owl:hasKey ( :hasSchoolDistrict
```

```
                          ) .


###  http://sws.ifi.uio.no/vocab/dsf#Time

:Time rdf:type owl:Class ;

      rdfs:subClassOf [ rdf:type owl:Restriction ;
                        owl:onProperty :timeIsRegistrationTimeForRelocationNotice ;
                        owl:allValuesFrom :RelocationNotice
                      ] ,
                      [ rdf:type owl:Restriction ;
                        owl:onProperty :timeIsRelocationTimeForRelocationNotice ;
                        owl:allValuesFrom :RelocationNotice
                      ] ,
                      [ rdf:type owl:Restriction ;
                        owl:onProperty :timeIsReloactionTimeForRelocation ;
                        owl:allValuesFrom :Relocation
                      ] ;

      owl:hasKey ( :hasTime
                 ) .



###  http://sws.ifi.uio.no/vocab/dsf#Year

:Year rdf:type owl:Class ;

      owl:hasKey ( :hasYear
                 ) .




#################################################################
#
#    General axioms
#
#################################################################


[ rdf:type owl:AllDisjointClasses ;
  owl:members ( :Address
                :ChurchParishes
                :Country
                :DateOfBirth
                :DayOfMonth
                :ElectionDistrict
                :FirstName
                :Gender
                :Identification
                :LastName
                :MaritalStatus
                :MiddleName
                :Month
                :PartOfTown
                :Person
                :PlaceOfBirth
```

```
                :Relocation
                :RelocationNotice
                :SchoolDistrict
                :Time
                :Year
              )
] .



###  Generated by the OWL API (version 3.2.3.22702) http://owlapi.sourceforge.net
```

Appendix D

The Mapping File for D2RQ

```
@prefix map: <http://sws.ifi.uio.no/project/dsf/charllo/mapping.n3#> .
@prefix db: <> .
@prefix dsf: <http://sws.ifi.uio.no/vocab/dsf#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix d2rq: <http://www.wiwiss.fu-berlin.de/suhl/bizer/D2RQ/0.1#> .
@prefix d2r: <http://sites.wiwiss.fu-berlin.de/suhl/bizer/d2r-server/config.rdf#> .
@prefix jdbc: <http://d2rq.org/terms/jdbc/> .

map:database a d2rq:Database;
        d2rq:jdbcDriver "com.mysql.jdbc.Driver";
        d2rq:jdbcDSN "jdbc:mysql://mysql.ifi.uio.no/charllo";
        d2rq:username "charllo";
        d2rq:password "dbCharllo";
        jdbc:autoReconnect "true";
        jdbc:zeroDateTimeBehavior "convertToNull";
        .

<> a d2r:Server;
    rdfs:label "DSF | sws.ifi.uio.no";
   d2r:baseURI <http://sws.ifi.uio.no/dsf/>;
    d2r:documentMetadata [
        rdfs:comment "No comment.";
    ];
        d2r:vocabularyIncludeInstances true;
    .


# Table Address
map:Address a d2rq:ClassMap;
        d2rq:dataStorage map:database;
        d2rq:uriPattern "Address/@@Address.addr|urlify@@";
        d2rq:class dsf:Address;
        d2rq:classDefinitionLabel "Address";
        .
map:Address__label a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:Address;
        d2rq:property rdfs:label;
        d2rq:column "Address.addr";
        d2rq:datatype xsd:string;
        .
map:Address_person a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:Address;
        d2rq:property dsf:addressAssociatedWithPerson;
        d2rq:propertyDefinitionLabel "Address person";
        d2rq:join "Address.addr <= Person.addr";
        d2rq:uriPattern "Person/@@Person.pid|urlify@@";
        .
map:Address_country a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:Address;
        d2rq:property dsf:addressLiesInCountry;
        d2rq:propertyDefinitionLabel "Address country";
        d2rq:join "Address.country => Country.country";
        d2rq:uriPattern "Country/@@Country.country|urlify@@";
        .
map:Address_electionDistrict a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:Address;
        d2rq:property dsf:addressBelongsToElectionDistrict;
        d2rq:propertyDefinitionLabel "Address electionDistrict";
        d2rq:join "Address.electionDistrict => ElectionDistrict.electionDistrict";
```

```
        d2rq:uriPattern "ElectionDistrict/@@ElectionDistrict.electionDistrict|urlify@@";
        .
map:Address_churchParish a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:Address;
        d2rq:property dsf:addressLiesInChurchParishes;
        d2rq:propertyDefinitionLabel "Address churchParish";
        d2rq:join "Address.churchParish => ChurchParishes.church";
        d2rq:uriPattern "ChurchParishes/@@ChurchParishes.church|urlify@@";
        .
map:Address_partOfTown a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:Address;
        d2rq:property dsf:addressLiesInPartOfTown;
        d2rq:propertyDefinitionLabel "Address partOfTown";
        d2rq:join "Address.partOfTown => PartOfTown.partOfTown";
        d2rq:uriPattern "PartOfTown/@@PartOfTown.partOfTown|urlify@@";
        .
map:Address_schoolDistrict a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:Address;
        d2rq:property dsf:addressLiesInSchoolDistrict;
        d2rq:propertyDefinitionLabel "Address schoolDistrict";
        d2rq:join "Address.schoolDistrict => SchoolDistrict.schoolDistrict";
        d2rq:uriPattern "SchoolDistrict/@@SchoolDistrict.schoolDistrict|urlify@@";
        .

# Table ChurchParishes
map:ChurchParishes a d2rq:ClassMap;
        d2rq:dataStorage map:database;
        d2rq:uriPattern "ChurchParishes/@@ChurchParishes.church|urlify@@";
        d2rq:class dsf:ChurchParishes;
        d2rq:classDefinitionLabel "ChurchParishes";
        .
map:ChurchParishes__label a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:ChurchParishes;
        d2rq:property rdfs:label;
        d2rq:column "ChurchParishes.church";
        d2rq:datatype xsd:string;
        .
map:ChurchParishes_address a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:ChurchParishes;
        d2rq:property dsf:churchParishesForAddress;
        d2rq:propertyDefinitionLabel "ChurchParishes address";
        d2rq:join "ChurchParishes.church <= Address.churchParish";
        d2rq:uriPattern "Address/@@Address.addr|urlify@@";
        .


# Table Country
map:Country a d2rq:ClassMap;
        d2rq:dataStorage map:database;
        d2rq:uriPattern "Country/@@Country.country|urlify@@";
        d2rq:class dsf:Country;
        d2rq:classDefinitionLabel "Country";
        .
map:Country__label a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:Country;
        d2rq:property rdfs:label;
        d2rq:column "Country.country";
        d2rq:datatype xsd:string;
        .
map:Country_address a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:Country;
```

```
        d2rq:property dsf:countryForAddress;
        d2rq:propertyDefinitionLabel "Country address";
        d2rq:join "Country.country <= Address.country";
        d2rq:uriPattern "Address/@@Address.addr|urlify@@";
        .

# Table DateOfBirth
map:DateOfBirth a d2rq:ClassMap;
        d2rq:dataStorage map:database;
        d2rq:uriPattern "DateOfBirth/@@DateOfBirth.day|urlify@@/@@DateOfBirth.month|
urlify@@/@@DateOfBirth.year|urlify@@";
        d2rq:class dsf:DateOfBirth;
        d2rq:classDefinitionLabel "DateOfBirth";
        .
map:DateOfBirth__label a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:DateOfBirth;
        d2rq:property rdfs:label;
        d2rq:pattern "@@DateOfBirth.day@@/@@DateOfBirth.month@@/@@DateOfBirth.year@@";
        d2rq:datatype xsd:string;
        .
map:DateOfBirth_year a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:DateOfBirth;
        d2rq:property dsf:dateOfBirthContainsYear;
        d2rq:propertyDefinitionLabel "DateOfBirth year";
        d2rq:column "DateOfBirth.year";
        d2rq:datatype xsd:string;
        .
map:DateOfBirth_month a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:DateOfBirth;
        d2rq:property dsf:dateOfBirthContainsMonth;
        d2rq:propertyDefinitionLabel "DateOfBirth month";
        d2rq:column "DateOfBirth.month";
        d2rq:datatype xsd:string;
        .
map:DateOfBirth_day a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:DateOfBirth;
        d2rq:property dsf:dateOfBirthContainsDay;
        d2rq:propertyDefinitionLabel "DateOfBirth day";
        d2rq:column "DateOfBirth.day";
        d2rq:datatype xsd:string;
        .
map:DateOfBirth_person a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:DateOfBirth;
        d2rq:property dsf:isDateOfBirthForPerson;
        d2rq:propertyDefinitionLabel "DateOfBirth person";
        d2rq:join "DateOfBirth.day <= Person.day";
        d2rq:join "DateOfBirth.month <= Person.month";
        d2rq:join "DateOfBirth.year <= Person.year";
        d2rq:uriPattern "Person/@@Person.pid|urlify@@";
        .
# Table DayOfMonth
map:DayOfMonth a d2rq:ClassMap;
        d2rq:dataStorage map:database;
        d2rq:uriPattern "DayOfMonth/@@DayOfMonth.day|urlify@@";
        d2rq:class dsf:DayOfMonth;
        d2rq:classDefinitionLabel "DayOfMonth";
        .
map:DayOfMonth__label a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:DayOfMonth;
        d2rq:property rdfs:label;
        d2rq:column "DayOfMonth.day";
```

```
        d2rq:datatype xsd:string;
        .
map:DayOfMonth_dob a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:DayOfMonth;
        d2rq:property dsf:dayIsPartOfDateOfBirth;
        d2rq:propertyDefinitionLabel "DayOfMonth dob";
        d2rq:join "DayOfMonth.day <= DateOfBirth.day";
        d2rq:uriPattern "DateOfBirth/@@DateOfBirth.day|urlify@@/@@DateOfBirth.month|
urlify@@/@@DateOfBirth.year|urlify@@";
        .

# Table ElectionDistrict
map:ElectionDistrict a d2rq:ClassMap;
        d2rq:dataStorage map:database;
        d2rq:uriPattern "ElectionDistrict/@@ElectionDistrict.electionDistrict|urlify@@";
        d2rq:class dsf:ElectionDistrict;
        d2rq:classDefinitionLabel "ElectionDistrict";
        .
map:ElectionDistrict__label a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:ElectionDistrict;
        d2rq:property rdfs:label;
        d2rq:column "ElectionDistrict.electionDistrict";
        d2rq:datatype xsd:string;
        .
map:ElectionDistrict_address a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:ElectionDistrict;
        d2rq:property dsf:electionDistrictForAddress;
        d2rq:propertyDefinitionLabel "ElectionDistrict address";
        d2rq:join "ElectionDistrict.electionDistrict <= Address.electionDistrict";
        d2rq:uriPattern "Address/@@Address.addr|urlify@@";
        .

# Table FirstName
map:FirstName a d2rq:ClassMap;
        d2rq:dataStorage map:database;
        d2rq:uriPattern "FirstName/@@FirstName.firstname|urlify@@";
        d2rq:class dsf:FirstName;
        d2rq:classDefinitionLabel "FirstName";
        .
map:FirstName__label a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:FirstName;
        d2rq:property rdfs:label;
        d2rq:column "FirstName.firstname";
        d2rq:datatype xsd:string;
        .
map:FirstName_person a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:FirstName;
        d2rq:property dsf:isFirstNameForPerson;
        d2rq:propertyDefinitionLabel "FirstName person";
        d2rq:join "FirstName.firstname <= Person_FirstName.firstname";
        d2rq:join "Person_FirstName.pid => Person.pid";
        d2rq:uriPattern "Person/@@Person.pid|urlify@@";
        .

# Table Gender
map:Gender a d2rq:ClassMap;
        d2rq:dataStorage map:database;
        d2rq:uriPattern "Gender/@@Gender.gender|urlify@@";
        d2rq:class dsf:Gender;
        d2rq:classDefinitionLabel "Gender";
        .
```

```
map:Gender__label a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:Gender;
        d2rq:property rdfs:label;
        d2rq:column "Gender.gender";
        d2rq:datatype xsd:string;
        .
map:Gender_person a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:Gender;
        d2rq:property dsf:isGenderForPerson;
        d2rq:propertyDefinitionLabel "Gender person";
        d2rq:join "Gender.gender <= Person.gender" ;
        d2rq:uriPattern "Person/@@Person.pid|urlify@@";
        .

# Table Identification
map:Identification a d2rq:ClassMap;
        d2rq:dataStorage map:database;
        d2rq:uriPattern "Identification/@@Identification.pid|urlify@@";
        d2rq:class dsf:Identification;
        d2rq:classDefinitionLabel "Identification";
        .
map:Identification__label a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:Identification;
        d2rq:property rdfs:label;
        d2rq:column "Identification.pid";
        d2rq:datatype xsd:string;
        .
map:Identification_person a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:Identification;
        d2rq:property dsf:isIdForPerson;
        d2rq:propertyDefinitionLabel "Identification person";
        d2rq:join "Identification.pid <= Person.pid" ;
        d2rq:uriPattern "Person/@@Person.pid|urlify@@";
        .

# Table LastName
map:LastName a d2rq:ClassMap;
        d2rq:dataStorage map:database;
        d2rq:uriPattern "LastName/@@LastName.lastname|urlify@@";
        d2rq:class dsf:LastName;
        d2rq:classDefinitionLabel "LastName";
        .
map:LastName__label a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:LastName;
        d2rq:property rdfs:label;
        d2rq:column "LastName.lastname";
        d2rq:datatype xsd:string;
        .
map:LastName_person a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:LastName;
        d2rq:property dsf:isLastNameForPerson;
        d2rq:propertyDefinitionLabel "LastName person";
        d2rq:join "LastName.lastname <= Person.lastname";
        d2rq:uriPattern "Person/@@Person.pid|urlify@@";
        .

# Table MaritalStatus
map:MaritalStatus a d2rq:ClassMap;
        d2rq:dataStorage map:database;
        d2rq:uriPattern "MaritalStatus/@@MaritalStatus.maritalStatus|urlify@@";
        d2rq:class dsf:MaritalStatus;
```

```
        d2rq:classDefinitionLabel "MaritalStatus";
        .
map:MaritalStatus__label a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:MaritalStatus;
        d2rq:property rdfs:label;
        d2rq:column "MaritalStatus.maritalStatus";
        d2rq:datatype xsd:string;
        .
map:MaritalStatus_person a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:MaritalStatus;
        d2rq:property dsf:isMaritalStatusForPerson;
        d2rq:propertyDefinitionLabel "MaritalStatus person";
        d2rq:join "MaritalStatus.maritalStatus <= Person.maritalStatus" ;
        d2rq:uriPattern "Person/@@Person.pid|urlify@@";
        .

# Table MiddleName
map:MiddleName a d2rq:ClassMap;
        d2rq:dataStorage map:database;
        d2rq:uriPattern "MiddleName/@@MiddleName.middlename|urlify@@";
        d2rq:class dsf:MiddleName;
        d2rq:classDefinitionLabel "MiddleName";
        .
map:MiddleName__label a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:MiddleName;
        d2rq:property rdfs:label;
        d2rq:column "MiddleName.middlename";
        d2rq:datatype xsd:string;
        .
map:MiddleName_person a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:MiddleName;
        d2rq:property dsf:isMiddleNameForPerson;
        d2rq:propertyDefinitionLabel "MiddleName person";
        d2rq:join "MiddleName.middlename <= Person_MiddleName.middlename";
        d2rq:join "Person_MiddleName.pid => Person.pid ";
        d2rq:uriPattern "Person/@@Person.pid|urlify@@";
        .

# Table Month
map:Month a d2rq:ClassMap;
        d2rq:dataStorage map:database;
        d2rq:uriPattern "Month/@@Month.month|urlify@@";
        d2rq:class dsf:Month;
        d2rq:classDefinitionLabel "Month";
        .
map:Month__label a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:Month;
        d2rq:property rdfs:label;
        d2rq:column "Month.month";
        d2rq:datatype xsd:string;
        .
map:Month_dob a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:Month;
        d2rq:property dsf:monthIsPartOfDateOfBirth;
        d2rq:propertyDefinitionLabel "Month dob";
        d2rq:join "Month.month <= DateOfBirth.month";
        d2rq:uriPattern "DateOfBirth/@@DateOfBirth.day|urlify@@/@@DateOfBirth.month|
urlify@@/@@DateOfBirth.year|urlify@@";
        .

# Table PartOfTown
```

```
map:PartOfTown a d2rq:ClassMap;
        d2rq:dataStorage map:database;
        d2rq:uriPattern "PartOfTown/@@PartOfTown.partOfTown|urlify@@";
        d2rq:class dsf:PartOfTown;
        d2rq:classDefinitionLabel "PartOfTown";
        .
map:PartOfTown__label a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:PartOfTown;
        d2rq:property rdfs:label;
        d2rq:column "PartOfTown.partOfTown";
        d2rq:datatype xsd:string;
        .
map:PartOfTown_address a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:PartOfTown;
        d2rq:property dsf:partOfTownForAddress;
        d2rq:propertyDefinitionLabel "PartOfTown address";
        d2rq:join "PartOfTown.partOfTown <= Address.partOfTown";
        d2rq:uriPattern "Address/@@Address.addr|urlify@@";
        .

# Table Person
map:Person a d2rq:ClassMap;
        d2rq:dataStorage map:database;
        d2rq:uriPattern "Person/@@Person.pid|urlify@@";
        d2rq:class dsf:Person;
        d2rq:classDefinitionLabel "Person";
        .
map:Person__label a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:Person;
        d2rq:property rdfs:label;
        d2rq:column "Person.pid";
        d2rq:datatype xsd:string;
        .
map:Person_pid a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:Person;
        d2rq:property dsf:personhasId;
        d2rq:propertyDefinitionLabel "Person pid";
        d2rq:join "Person.pid => Identification.pid ";
        d2rq:uriPattern "Identification/@@Identification.pid|urlify@@";
        .
map:Person_pidMother a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:Person;
        d2rq:property dsf:personIsMotherFor;
        d2rq:propertyDefinitionLabel "Person pid Mother";
        d2rq:join "Person.pid <= Child.mother";
        d2rq:uriPattern "Person/@@Child.pid|urlify@@";
        d2rq:alias "Person AS Child";
        .
map:Person_pidFather a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:Person;
        d2rq:property dsf:personIsFatherFor;
        d2rq:propertyDefinitionLabel "Person pid Father";
        d2rq:join "Person.pid <= Child.father";
        d2rq:uriPattern "Person/@@Child.pid|urlify@@";
        d2rq:alias "Person AS Child";
        .
map:Person_pidCoMother a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:Person;
        d2rq:property dsf:personIsCoMotherFor;
        d2rq:propertyDefinitionLabel "Person pid CoMother";
        d2rq:join "Person.pid <= Child.comother";
```

```
        d2rq:uriPattern "Person/@@Child.pid|urlify@@";
        d2rq:alias "Person AS Child";
        .
map:Person_spouse a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:Person;
        d2rq:property dsf:personIsMarriedTo;
        d2rq:propertyDefinitionLabel "Person spouse";
        d2rq:join "Person.pid <= Marriage.isMarriedTo1";
        d2rq:join "Marriage.isMarriedTo2 => Spouse.pid";
        d2rq:uriPattern "Person/@@Spouse.pid|urlify@@";
        d2rq:alias "Person AS Spouse";
        .
map:Person_pidRelocation a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:Person;
        d2rq:property dsf:personIsRelocating;
        d2rq:propertyDefinitionLabel "Person pid Relocation";
        d2rq:join "Person.pid <= Relocation.pid";
        d2rq:uriPattern "Relocation/@@Relocation.number|urlify@@";
        .
map:Person_dob a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:Person;
        d2rq:property dsf:personHasDateOfBirth;
        d2rq:propertyDefinitionLabel "Person dob";
        d2rq:join "Person.year => DateOfBirth.year";
        d2rq:join "Person.month => DateOfBirth.month";
        d2rq:join "Person.day => DateOfBirth.day";
        d2rq:uriPattern "DateOfBirth/@@DateOfBirth.day|urlify@@/@@DateOfBirth.month|
urlify@@/@@DateOfBirth.year|urlify@@";
        .
map:Person_gender a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:Person;
        d2rq:property dsf:personHasGender;
        d2rq:propertyDefinitionLabel "Person gender";
        d2rq:join "Person.gender => Gender.gender";
        d2rq:uriPattern "Gender/@@Gender.gender|urlify@@";
        .
map:Person_lastname a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:Person;
        d2rq:property dsf:personHasLastName;
        d2rq:propertyDefinitionLabel "Person lastname";
        d2rq:join "Person.lastname => LastName.lastname";
        d2rq:uriPattern "LastName/@@LastName.lastname|urlify@@";
        .
map:Person_maritalStatus a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:Person;
        d2rq:property dsf:personHasMaritalStatus;
        d2rq:propertyDefinitionLabel "Person maritalStatus";
        d2rq:join "Person.maritalStatus => MaritalStatus.maritalStatus";
        d2rq:uriPattern "MaritalStatus/@@MaritalStatus.maritalStatus|urlify@@";
        .
map:Person_placeOfBirth a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:Person;
        d2rq:property dsf:personHasPlaceOfBirth;
        d2rq:propertyDefinitionLabel "Person placeOfBirth";
        d2rq:join "Person.placeOfBirth => PlaceOfBirth.placeOfBirth";
        d2rq:uriPattern "PlaceOfBirth/@@PlaceOfBirth.placeOfBirth|urlify@@";
        .
map:Person_addr a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:Person;
        d2rq:property dsf:personLivesInAddress;
        d2rq:propertyDefinitionLabel "Person addr";
```

```
        d2rq:join "Person.addr => Address.addr";
        d2rq:uriPattern "Address/@@Address.addr|urlify@@";
        .
map:Person_comother a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:Person;
        d2rq:property dsf:personHasCoMother;
        d2rq:propertyDefinitionLabel "Person comother";
        d2rq:join "Person.comother => Comother.pid";
        d2rq:uriPattern "Person/@@Comother.pid|urlify@@";
        d2rq:alias "Person AS Comother";
        .
map:Person_father a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:Person;
        d2rq:property dsf:personHasFather;
        d2rq:propertyDefinitionLabel "Person father";
        d2rq:join "Person.father => Father.pid";
        d2rq:uriPattern "Person/@@Father.pid|urlify@@";
        d2rq:alias "Person AS Father";
        .
map:Person_mother a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:Person;
        d2rq:property dsf:personHasMother;
        d2rq:propertyDefinitionLabel "Person mother";
        d2rq:join "Person.mother => Mother.pid";
        d2rq:uriPattern "Person/@@Mother.pid|urlify@@";
        d2rq:alias "Person AS Mother";
        .
map:Person_FirstName a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:Person;
        d2rq:property dsf:personHasFirstName;
        d2rq:propertyDefinitionLabel "Person_FirstName pid";
        d2rq:join "Person.pid <= Person_FirstName.pid";
        d2rq:join "Person_FirstName.firstname => FirstName.firstname";
        d2rq:uriPattern "FirstName/@@FirstName.firstname|urlify@@";
        .
map:Person_MiddleName a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:Person;
        d2rq:property dsf:personHasMiddleName;
        d2rq:propertyDefinitionLabel "Person_MiddleName pid";
        d2rq:join "Person.pid <= Person_MiddleName.pid";
        d2rq:join "Person_MiddleName.middlename => MiddleName.middlename";
        d2rq:uriPattern "MiddleName/@@MiddleName.middlename|urlify@@";
        .

# Table PlaceOfBirth
map:PlaceOfBirth a d2rq:ClassMap;
        d2rq:dataStorage map:database;
        d2rq:uriPattern "PlaceOfBirth/@@PlaceOfBirth.placeOfBirth|urlify@@";
        d2rq:class dsf:PlaceOfBirth;
        d2rq:classDefinitionLabel "PlaceOfBirth";
        .
map:PlaceOfBirth__label a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:PlaceOfBirth;
        d2rq:property rdfs:label;
        d2rq:column "PlaceOfBirth.placeOfBirth";
        d2rq:datatype xsd:string;
        .
map:PlaceOfBirth_person a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:PlaceOfBirth;
        d2rq:property dsf:isPlaceOfBirthForPerson;
        d2rq:propertyDefinitionLabel "PlaceOfBirth person";
```

```
        d2rq:join "PlaceOfBirth.placeOfBirth <= Person.placeOfBirth" ;
        d2rq:uriPattern "Person/@@Person.pid|urlify@@";
        .

# Table Relocation
map:Relocation a d2rq:ClassMap;
        d2rq:dataStorage map:database;
        d2rq:uriPattern "Relocation/@@Relocation.number|urlify@@";
        d2rq:class dsf:Relocation;
        d2rq:classDefinitionLabel "Relocation";
        .
map:Relocation__label a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:Relocation;
        d2rq:property rdfs:label;
        d2rq:column "Relocation.number";
        d2rq:datatype xsd:string;
        .
map:Relocation_pid a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:Relocation;
        d2rq:property dsf:relocationConcernsPerson;
        d2rq:propertyDefinitionLabel "Relocation pid";
        d2rq:join "Relocation.pid => Person.pid" ;
        d2rq:uriPattern "Person/@@Person.pid|urlify@@";
        .
map:Relocation_relocationDate a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:Relocation;
        d2rq:property dsf:relocationHappendAtTime;
        d2rq:propertyDefinitionLabel "Relocation relocationDate";
        d2rq:join "Relocation.relocationDate => Time.date" ;
        d2rq:uriPattern "Time/@@Time.date|urlify@@";
        .
map:Relocation_relocationNotice a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:Relocation;
        d2rq:property dsf:relocationIsReportedInRelocationNotice;
        d2rq:propertyDefinitionLabel "Relocation relocationNotice";
        d2rq:join "Relocation.relocationNotice => RelocationNotice.number";
        d2rq:uriPattern "RelocationNotice/@@RelocationNotice.number|urlify@@";
        .

# Table RelocationNotice
map:RelocationNotice a d2rq:ClassMap;
        d2rq:dataStorage map:database;
        d2rq:uriPattern "RelocationNotice/@@RelocationNotice.number|urlify@@";
        d2rq:class dsf:RelocationNotice;
        d2rq:classDefinitionLabel "RelocationNotice";
        .
map:RelocationNotice__label a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:RelocationNotice;
        d2rq:property rdfs:label;
        d2rq:column "RelocationNotice.number";
        d2rq:datatype xsd:string;
        .
map:RelocationNotice_movingAwayAddr a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:RelocationNotice;
        d2rq:property dsf:relocationNoticeHasMovingAwayAddress;
        d2rq:propertyDefinitionLabel "RelocationNotice movingAwayAddr";
        d2rq:join "RelocationNotice.movingAwayAddr => Address.addr";
        d2rq:uriPattern "Address/@@Address.addr|urlify@@";
        .
map:RelocationNotice_movingToAddr a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:RelocationNotice;
```

```
        d2rq:property dsf:relocationNoticeHasMovingToAddress;
        d2rq:propertyDefinitionLabel "RelocationNotice movingToAddr";
        d2rq:join "RelocationNotice.movingToAddr => Address.addr";
        d2rq:uriPattern "Address/@@Address.addr|urlify@@";
        .
map:RelocationNotice_registrationDate a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:RelocationNotice;
        d2rq:property dsf:relocationNoticeHasRegistrationTime;
        d2rq:propertyDefinitionLabel "RelocationNotice registrationDate";
        d2rq:join "RelocationNotice.registrationDate => Time.date" ;
        d2rq:uriPattern "Time/@@Time.date|urlify@@";
        .
map:RelocationNotice_relocationDate a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:RelocationNotice;
        d2rq:property dsf:relocationNoticeConcernsRelocationTime;
        d2rq:propertyDefinitionLabel "RelocationNotice relocationDate";
        d2rq:join "RelocationNotice.relocationDate => Time.date" ;
        d2rq:uriPattern "Time/@@Time.date|urlify@@";
        .

# Table SchoolDistrict
map:SchoolDistrict a d2rq:ClassMap;
        d2rq:dataStorage map:database;
        d2rq:uriPattern "SchoolDistrict/@@SchoolDistrict.schoolDistrict|urlify@@";
        d2rq:class dsf:SchoolDistrict;
        d2rq:classDefinitionLabel "SchoolDistrict";
        .
map:SchoolDistrict__label a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:SchoolDistrict;
        d2rq:property rdfs:label;
        d2rq:column "SchoolDistrict.schoolDistrict";
        d2rq:datatype xsd:string;
        .
map:SchoolDistrict_address a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:SchoolDistrict;
        d2rq:property dsf:schoolDistrictForAddress;
        d2rq:propertyDefinitionLabel "SchoolDistrict address";
        d2rq:join "SchoolDistrict.schoolDistrict <= Address.schoolDistrict";
        d2rq:uriPattern "Address/@@Address.addr|urlify@@";
        .

# Table Time
map:Time a d2rq:ClassMap;
        d2rq:dataStorage map:database;
        d2rq:uriPattern "Time/@@Time.date|urlify@@";
        d2rq:class dsf:Time;
        d2rq:classDefinitionLabel "Time";
        .
map:Time__label a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:Time;
        d2rq:property rdfs:label;
        d2rq:column "Time.date";
        d2rq:datatype xsd:string;
        .
map:Time_dateRelocationTime a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:Time;
        d2rq:property dsf:timeIsRelocationTimeForRelocationNotice;
        d2rq:propertyDefinitionLabel "Time RelocationNotice";
        d2rq:join "Time.date <= Relocation.relocationDate";
        d2rq:uriPattern "RelocationNotice/@@RelocationNotice.number|urlify@@";
        .
```

```
map:Time_dateRegistrationTime a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:Time;
        d2rq:property dsf:timeIsRegistrationTimeForRelocationNotice;
        d2rq:propertyDefinitionLabel "Time dateRegistration";
        d2rq:join "Time.date <= RelocationNotice.registrationDate";
        d2rq:uriPattern "RelocationNotice/@@RelocationNotice.number|urlify@@";
        .
map:Time_dateRelocationTime2 a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:Time;
        d2rq:property dsf:timeIsReloactionTimeForRelocation;
        d2rq:propertyDefinitionLabel "Time dateRelocation";
        d2rq:join "Time.date <= Relocation.relocationDate";
        d2rq:uriPattern "Relocation/@@Relocation.number|urlify@@";
        .

# Table Year
map:Year a d2rq:ClassMap;
        d2rq:dataStorage map:database;
        d2rq:uriPattern "Year/@@Year.year|urlify@@";
        d2rq:class dsf:Year;
        d2rq:classDefinitionLabel "Year";
        .
map:Year__label a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:Year;
        d2rq:property rdfs:label;
        d2rq:column "Year.year";
        d2rq:datatype xsd:string;
        .
map:Year_dob a d2rq:PropertyBridge;
        d2rq:belongsToClassMap map:Year;
        d2rq:property dsf:yearIsPartOfDateOfBirth;
        d2rq:propertyDefinitionLabel "Year dob";
        d2rq:join "Year.year <= DateOfBirth.year";
        d2rq:uriPattern "DateOfBirth/@@DateOfBirth.day|urlify@@/@@DateOfBirth.month|
urlify@@/@@DateOfBirth.year|urlify@@";
        .
```

Bibliography

Bechhofer, S., Van Harmelen, F., Hendler, J., Horrocks, I., McGuinness, D., Patel-Schneider, P., et al. (2004). Owl web ontology language reference. *W3C recommendation*, *10*, 2006–01.

Bizer, C., & Cyganiak, R. (2006). *D2r server-publishing relational databases on the semantic web.*

Bizer, C., & Cyganiak, R. (2007). D2rq-lessons learned. In *W3c workshop on rdf access to relational databases.*

Bizer, C., Cyganiak, R., Garbers, J., Maresch, O., & Becker, C. (2011, May). *The D2RQ Platform v0.7 - Treating Non-RDF Relational Databases as Virtual RDF Graphs.* `http://www4.wiwiss.fu-berlin.de/bizer/d2rq/spec/`.

Blace, R. (2009, June). *OWL 2 in Action - Property Chains.* `http://semwebprogramming.org/?p=175`.

Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., Poggi, A., Rodriguez-Muro, M., et al. (2009). Ontologies and databases: The DL-Lite approach. *Reasoning Web. Semantic Technologies for Information Systems*, 255–356.

Cerbah, F. (2008). Learning highly structured semantic repositories from relational databases. *The Semantic Web: Research and Applications*, 777–781.

Cyganiak, R., & Bizer, C. (2006). D2R Server: A Semantic Web front-end to existing relational databases. *XML Tage*, 171–173.

D2RQ/Update and D2R/Update Server. (2011). `http://d2rqupdate.cs.technion.ac.il/`.

Davies, J., Fensel, D., & Van Harmelen, F. (2003). *Towards the semantic web*. Wiley Online Library.

Editing OWL Ontologies. (2011, October). `http://www.altova.com/semanticworks/owl-editor.html`.

Edraw Soft - Vector-Based Graphic Design. (2011, October). `http://www.edrawsoft.com/ORM-Diagram.php`.

Folkeregisteret. (2011, February). `http://www.skatteetaten.no/Alt-om/Folkeregistrering/`.

Halpin, T., Morgan, A., & Morgan, T. (2008). *Information modeling and relational databases*. Morgan Kaufmann.

Hebeler, J., Fisher, M., Blace, R., & Perez-Lopez, A. (2009). *Semantic web programming*. Wiley Publishing.

Hert, M., Reif, G., & Gall, H. (2010). Updating relational data via sparql/update. In *Proceedings of the 2010 edbt/icdt workshops* (p. 24).

Hitzler, P., Krötzsch, M., & Rudolph, S. (2009). *Foundations of semantic web technologies*. Chapman & Hall/CRC.

Hodrob, R., & Jarrar, M. (2010). Mapping orm into owl 2. In *Proceedings of the 1st international conference on intelligent semantic web-services and applications* (pp. 9:1–9:7). ACM.

IT Liberator. (2011). `http://www.idedata.no/firma/losninger/it.htm`.

Jarrar, M. (2005). Towards methodological principles for ontology engineering. *STARLAB. Vrije Universiteit Brussel, Brussel*.

Jarrar, M. (2007). *Mapping ORM into the SHOIN/OWL description logic: towards a methodological and expressive graphical notation for ontology engineering.*

Keet, C. (2007). Mapping the Object-Role Modeling language ORM2 into Description Logic language DLRifd. *Arxiv preprint cs/0702089*.

Konstantinou, N., Spanos, D., Chalas, M., Solidakis, E., & Mitrou, N. (2006). *VisAVis: An approach to an intermediate layer between ontologies and relational database contents.*

Laclavik, M. (2007). Rdb2onto: relational database data to ontology individuals mapping. In *Proceeding of ninth international conference of informatics.*

Lee, T., Hendler, J., Lassila, O., et al. (2001). The semantic web. *Scientific American*, *284*(5), 34–43.

McGuinness, D., Van Harmelen, F., et al. (2004). Owl web ontology language overview. *W3C recommendation*, *10*, 2004–03.

NORMA. (2011, May). `http://www.objectrolemodeling.com/AboutORM/ORMTools/NORMA/tabid/87/Default.aspx`.

NORMA -The Software! (2011, May). `http://www.ormfoundation.org/files/folders/norma_the_software/category1449.aspx`.

OntoStudio. (2011, October). `http://www.ontoprise.de/en/products/ontostudio/`.

The ORM Foundation - ORM Lite. (2011, October). `http://www.ormfoundation.org/files/folders/orm_lite/entry2774.aspx`.

Pellet Integrity Constraints: Validating RDF with OWL. (2011). `http://clarkparsia.com/pellet/icv/`.

Protégé. (2011, May). `http://protege.stanford.edu/`.

Protégé OWL Tutorial. (2011, April). `http://owl.cs.manchester.ac.uk/tutorials/protegeowltutorial/`.

QuOnto. (2011, February). `http://semanticweb.org/wiki/QuOnto`.

QuOnto QUerying ONTOlogies. (2011, Februar). `http://www.dis.uniroma1.it/~quonto/`.

RDBToOnto: From Relational Databases to Ontologies. (2011, June). `http://www.tao-project.eu/researchanddevelopment/demosanddownloads/RDBToOnto.html`.

Robinson, R., Henricksen, K., & Indulska, J. (2007). Xcml: A runtime representation for the context modelling language. In *Proceedings of the fifth ieee international conference on pervasive computing and communications workshops* (pp. 20–26). IEEE Computer Society.

Sahoo, S., Halb, W., Hellmann, S., Idehen, K., Thibodeau Jr, T., Auer, S., et al. (2009). A survey of current approaches for mapping of relational databases to RDF. *W3C RDB2RDF Incubator Group report*.

Šeleng, M., Laclavík, M., Balogh, Z., & Hluchỳ, L. (2007). *RDB2Onto: Approach for creating semantic metadata from relational database data.*

Semicolon II. (2011). `http://www.semicolon.no/Hjemmeside-E.html`.

Skagestein, G. (1996). Dataorientert systemutvikling. *Universitetsforlaget, Oslo*.

Skagestein, G., & Normann, R. (2008). A closer look at the join-equality constraint. In *On the move to meaningful internet systems: Otm 2008 workshops* (pp. 780–786).

Skatteetaten. (2011, May). `http://www.skatteetaten.no/no/`.

Tool for Relational Data to Ontology Individuals Mapping (RDB2Onto tool). (2011, June). `http://nazou.fiit.stuba.sk/home/?page=rdb2onto`.

TopBraid Composer. (2011, October). `http://www.topquadrant.com/products/TB_Composer.html`.

Wagih, H., ElZanfaly, D., & Kouta, M. (2011). Mapping object role modeling 2 schemes to owl2 ontologies. In *Computer research and development (iccrd), 2011 3rd international conference on* (Vol. 3, pp. 126–132).

Xu, Z., Zhang, S., & Dong, Y. (2006). *Mapping between relational database schema and OWL ontology for deep annotation.*